

内 容 提 要

本书的核心是讲述 Object Pascal 和 VCL 的重、难点知识 (即所谓“精要”), 集中于第 3 章、第 5 章。第 2 章、第 4 章阐明相应的一些基本概念, 是为初学者而设立的。

本书也是一本参考手册, 包括了“IDE 的快捷键列表”(2.8 节)、“编译指令”(3.4 节)、“VCL 消息大全”(5.2.4 节)、“常用函数和过程”(第 8 章)以及一些常用的、重要的开发技巧(第 9 章)等内容。

本书同时也较为全面地讲述了组件开发知识, 集中于第 6 章、第 7 章。

第 10 章通过一个实例综合运用了全书内容。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有, 侵权必究。

图书在版编目(CIP)数据

Delphi 精要/罗小平编著. —北京: 电子工业出版社, 2004.1

(Borland In-Depth Series\Borland 大系)

ISBN 7-5053-9412-6

.D... 罗... 软件工具 - 程序设计 .TP311.56

中国版本图书馆 CIP 数据核字 (2003) 第 110153 号

责任编辑: 周 筠 陈元玉

技术编辑: 韩 磊

印 刷:

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

经 销: 各地新华书店

开 本: 787×980 1/16 印张: 25.25 字数: 600 千字

印 次: 2004 年 1 月第 1 次印刷

印 数: 6 000 册 定价: 39.8 元 (含光盘 1 张)

凡购买电子工业出版社的图书, 如有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系。联系电话: (010) 68279077。质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

目 录

第 1 章 关于 Delphi 与本书.....	(1)
1.1 Delphi 是什么	(1)
1.1.1 Delphi 的历史	(1)
1.1.2 选择 Delphi 意味着什么	(2)
1.1.3 如何使用 Delphi 编程	(3)
1.2 本书的特点	(4)
1.2.1 本书有哪些内容，没有哪些内容	(5)
1.2.2 本书是如何写作的	(5)
1.2.3 如何阅读本书	(5)
第 2 章 Object Pascal 入门.....	(7)
2.1 运算符	(7)
2.1.1 赋值运算符	(7)
2.1.2 比较运算符	(7)
2.1.3 逻辑运算符	(8)
2.1.4 算术运算符	(8)
2.1.5 按位运算符	(9)
2.1.6 其他运算符和模拟运算	(9)
2.2 常量和变量	(12)
2.2.1 无类型常量和有类型常量	(13)
2.2.2 全局变量和局部变量	(13)
2.2.3 声明时使用编译时函数	(14)
2.2.4 资源字符串	(15)
2.3 过程和函数	(15)
2.3.1 过程和函数的区别	(15)
2.3.2 子过程	(16)
2.4 数据类型	(17)
2.4.1 常用类型和复杂类型	(17)



2.4.2	不同语言的数据类型对照表	(17)
2.5	程序流程控制	(18)
2.5.1	条件分支	(18)
2.5.2	循环	(19)
2.5.3	跳转	(20)
2.5.4	用过程辅助实现流程控制	(21)
2.6	单元的组织结构	(24)
2.6.1	Program 的组织结构	(24)
2.6.2	Unit 的组织结构	(25)
2.6.3	单元循环引用	(27)
2.7	with...do 语句的用法	(28)
2.8	IDE 的快捷键列表	(29)
第 3 章 Object Pascal 精要		(31)
3.1	数据类型及其相互关系	(31)
3.1.1	数据类型概述	(32)
3.1.2	变量的内存分配和释放	(48)
3.1.3	数据的内存结构	(52)
3.1.4	强数据类型与类型转化	(53)
3.2	过程和函数	(59)
3.2.1	作用域	(60)
3.2.2	参数传递	(62)
3.2.3	声明指令	(66)
3.3	类和类成员	(66)
3.3.1	类和类成员概述	(67)
3.3.2	深入认识方法	(69)
3.3.3	深入认识属性	(78)
3.3.4	深入认识事件	(84)
3.3.5	类成员重新声明	(86)
3.3.6	inherited 释疑	(88)
3.3.7	接口的真相	(89)
3.4	编译指令	(95)
3.4.1	开关指令	(96)
3.4.2	参数指令	(99)

3.4.3 条件指令	(102)
第 4 章 VCL 入门	(104)
4.1 VCL 概述	(104)
4.2 组件与控件的概念	(105)
4.3 使用 VCL	(105)
4.4 扩展 VCL	(107)
第 5 章 VCL 精要	(108)
5.1 揭开 VCL 的神秘面纱	(108)
5.1.1 VCL 架构	(108)
5.1.2 构造和析构的内幕	(111)
5.1.3 虚拟方法表和动态方法表	(115)
5.1.4 TObject 如何使用虚拟方法表	(118)
5.1.5 运行时类型信息	(122)
5.2 VCL 的消息机制	(130)
5.2.1 VCL 消息机制	(130)
5.2.2 处理消息的八种方法	(134)
5.2.3 选用什么方法发送消息	(140)
5.2.4 VCL 消息大全	(144)
5.3 多态性	(161)
5.3.1 多态性的概念	(162)
5.3.2 多态性和虚方法的关系	(164)
第 6 章 组件开发实战	(167)
6.1 三种组件开发方法	(167)
6.1.1 继承、聚合和子类化	(167)
6.1.2 接口、虚方法和辅助类的选择	(169)
6.2 文件拖放监视器	(169)
6.2.1 文件拖放原理	(169)
6.2.2 文件拖放实例	(170)
6.2.3 组件封装	(171)
6.3 托盘组件	(175)
6.3.1 装入托盘图标	(175)
6.3.2 在应用程序最小化时去掉状态栏的图	(177)



6.3.3	给托盘图标增加接收鼠标消息功能	(179)
6.3.4	处理鼠标消息	(181)
6.3.5	显示动画图标	(182)
6.3.6	设置程序的自动启动功能	(182)
6.3.7	组件封装	(183)
6.4	自动下拉的 TComboBox	(190)
6.5	开发数据敏感控件	(195)
6.5.1	数据敏感原理	(196)
6.5.2	开发日期敏感控件	(196)
6.6	开发聚合组件	(203)
6.6.1	开发 LabelDBDatePicker	(203)
6.6.2	加强 LabelDBDatePicker	(206)
6.7	开发图形图像控件	(211)
6.8	开发 QuickReport 组件	(218)
第 7 章	组件开发相关工作	(225)
7.1	包和包编译指令	(225)
7.2	创建组件图标	(226)
7.3	属性编辑器	(227)
7.4	组件编辑器	(231)
第 8 章	常用函数和过程	(235)
8.1	数据类型转化类	(235)
8.1.1	数值和字符串的相互转化	(235)
8.1.2	整数和字符串的相互转化	(236)
8.1.3	实数和字符串的相互转化	(236)
8.1.4	实数子类型的相互转化	(237)
8.1.5	布尔类型和字符串的相互转化	(237)
8.2	字符串处理类	(237)
8.2.1	字符串的分类	(240)
8.2.2	和字符串相关的类	(241)
8.3	流处理类	(245)
8.4	内存管理、程序流程控制类	(250)
8.4.1	内存管理	(250)



8.4.2 程序流程控制	(251)
8.5 文件操作类	(252)
8.5.1 使用文件句柄进行 I/O 处理	(252)
8.5.2 使用 Pascal 文件变量进行 I/O 处理	(253)
8.5.3 面向对象文件 I/O 处理	(255)
8.5.4 文件属性操作	(255)
8.5.5 其他函数和方法	(256)
8.6 日期时间类	(258)
8.6.1 获取/合成日期/时间	(258)
8.6.2 日期/时间和字符串的转换	(259)
8.6.3 日期/时间的运算	(259)
8.7 VCL 类	(260)
8.7.1 Classes 单元	(260)
8.7.2 Controls 单元	(262)
8.7.3 Dialogs 单元	(262)
8.8 位运算类	(264)
8.9 图形图像类	(266)
第 9 章 高级开发技巧	(268)
9.1 自定义窗口过程	(268)
9.2 自定义消息及其替代方法	(270)
9.3 自定义系统惟一消息	(272)
9.4 新颖的类工厂	(275)
9.5 使用对象库	(281)
9.6 非发布 (published) 数据的持久化	(287)
9.7 使用回调函数	(288)
9.8 使用递归算法	(290)
9.9 编写 NT 服务程序	(294)
9.10 编写只能惟一运行的程序	(295)
9.11 字段类型全家福	(298)
9.12 获取数据库结构信息	(300)
9.13 深入使用 TCanvas	(301)
9.14 指针列表类的使用	(308)
9.15 结构化存储技术	(312)





9.16	挂钩技术	(321)
9.17	TRichEdit 高级开发	(327)
9.18	用 TTreeView 分析数据表的结构	(334)
9.19	SQL 语句分析器	(339)
9.20	剪贴板高级编程	(344)
第 10 章 综合例子——使用 Socket 传输多个文件		(350)
10.1	Socket 简介	(350)
10.2	TServerSocket 和 TClientSocket	(351)
10.3	设计通讯协议	(354)
10.4	实现服务端	(357)
10.5	实现客户端	(367)
10.6	组件封装	(374)
10.7	自动下载技术在项目中的应用	(391)



第 1 章 关于 Delphi 与本书

在一本书的开头，总应该说点什么，我也不能脱离俗套。既然是俗套，或许也的确是必要的。黑格尔不是说过：存在即合理。

既然要说，就得说点有价值的，如果全是废话、大话、空话，就难免遭人唾骂。

在本章里，我想谈谈两个方面的话题：

- (1) 我对 Delphi 的一些认识。
- (2) 这本书的特点。

1.1 Delphi 是什么

这一节希望读者朋友对 Delphi 有一个总体的认识，包括其发展历史、特点。无论是在网络论坛、办公室，还是在技术座谈会上，时常有人询问“Delphi 有没有前途”、“我该选择 .NET、Java 还是 Delphi”、“Delphi 的旗帜还能打多久”之类的问题，即使是正在使用 Delphi 的朋友，也常常被这样的问题困扰。本节不指望能正面回答这些问题，但是希望能给出一些理由让困惑的朋友们平静下来，从而将时间和精力放到更重要的事情上去。

1.1.1 Delphi 的历史

在 DOS 时代，程序员可选择的开发工具很少：要么是易用但低效的 BASIC 语言，要么是高效但难用的汇编语言。即使到了 Windows 3.X 时代，仍然是两难选择：要么是容易使用但功能十分有限的 Visual BASIC，要么是强大但难以使用的 C/C++。

但是到了 1995 年，情况发生了很大变化。那就是 Delphi 1.0 在 Borland 公司诞生了。Delphi 1.0 运行在 Windows 3.X 平台上，以 Object Pascal（面向对象的 Pascal，从 Pascal 发展而来）为开发语言，它提供了一种全新的 Windows 程序开发方法：可视化的开发环境、真编译后的可执行程序、DDL 和支持数据库软件开发。Delphi 1.0 是第一个综合了可视化开发技术、优化的源代码编译器和可扩展的数据库访问引擎的 Windows 平台开发工具。Delphi 1.0 奠定了 RAD（Rapid Application Development，快速应用程序开发）的概念。

尽管在 Delphi 3 开发前后，著名的 Delphi 首席设计师 Anders Hejlsberg（第一个 Turbo Pascal 编译器的作者）首席技术总裁 Paul Gross 离开 Borland 公司去了 Microsoft 公司，并且在整个的 Delphi 发展过程中，Borland 公司也出现了一些重大决策失误，但是 Delphi 仍然拥有全世界众多的爱好者和使用者。



8年过去了，Delphi 已经进入了 7.0 阶段，并且推出了 Linux 平台的版本 Kylix，Delphi 8 for.NET 也已经面世。

1.1.2 选择 Delphi 意味着什么

虽然很多人和我一样选择了 Delphi，但是有一点是肯定的，我们多半同时选择了或迟早要面临选择别的工具。

在 DOS 和 Windows 3.X 时代，可选择的工具太少，甚至于没有选择，所以大家只好安贫乐道，比如 Anders Hejlsberg、求伯君二位前辈用汇编语言分别完成了 Delphi 1.0 的编译器和 WPS。到了开发工具群雄并起的时候，面临众多诱惑和困惑，要作出让自己满意的选择反而成了难事。

我不打算说服你选择 Delphi，我只能说，选择自己最适用的工具。

Delphi 对于程序员来说，应该是一个工具。选择 Delphi，是因为在一定阶段里，它能和我们构成最佳匹配，而再没有别的原因。比如你受到身边 Delphi 高手的鼓动，经理指定 Delphi 为项目的开发工具，公司的主流开发工具就是 Delphi，为了技术研究目的选择 Delphi，等等。要是我说 Delphi 是最先进、功能最强大的开发工具，肯定很多人会跳起来说出 Delphi 的一大堆不足和其他工具的许多优势。网上有很多程序来测试 Delphi 和 VC、VB 等开发出来的可执行程序的运行效率，并指出某某比其他的要快出百分之多少、哪个传送消息要快得多，我想这些对于软件用户来说都没有多大价值。影响一个人选择的因素很多，要说哪个因素总是占绝对的优势地位，是很牵强的。

但是 Delphi 也的确有众多出类拔萃的优势，我最看重的是以下几点：

- (1) 编译速度非常快。
- (2) IDE 反应速度很快。
- (3) 完全开放的 VCL 源代码和规范简练的帮助系统。

选择开发工具，和选择恋人是一样的道理。你的选择对于你可能是最适合的，你认为他/她是全世界最好的，但是其他人或许就不这么看。因此，大可不必将精力过分专注于本节开头提到的那些问题。

慎重作出选择，然后忠实于自己的选择。如果朝三暮四，最后只能是什么都不能精通，浪费时间；当然对于那些一点就通的天才就另当别论。只有乐于此道的人，才能乐在其中；上天总是给不安分者更多的痛苦。

在大多数时候，众多语言都能达到殊途同归的效果，C/C++、Object Pascal、Java、C#有异曲同工之妙。如果一定要选择最好的，就得将它们全部精通了，才能作出绝对公正的决定，但是世界上恐怕没有这样的人；即使有，他也会非常忙，因为很多人都恳请他帮自己作出选择，他最终只好偷偷跑到深山里躲起来，否则他的电话会被打爆或者被提问者的唾沫星子淹死。

最终能作出什么成果，关键还是在于工具使用者。武林高手发出一声长啸，就可以让敌人丧失听觉，但是弱女子手提宝剑、面对身负重伤的敌手时却不见得稳操胜券。

所以，如果你选择了 Delphi，那么赶快去掌握它吧，本书将助你一臂之力；如果你还在彷徨中虚度光阴，那就尽早作出选择吧。



1.1.3 如何使用 Delphi 编程

Delphi 是一个快速开发工具，“快速”的特点主要体现在两个方面：

1. 提供众多的向导程序

Delphi 向导程序主要包含在对象库（Object Repository）中，选择菜单 File|New|Other 可以看到所有的向导程序，如图 1-1。

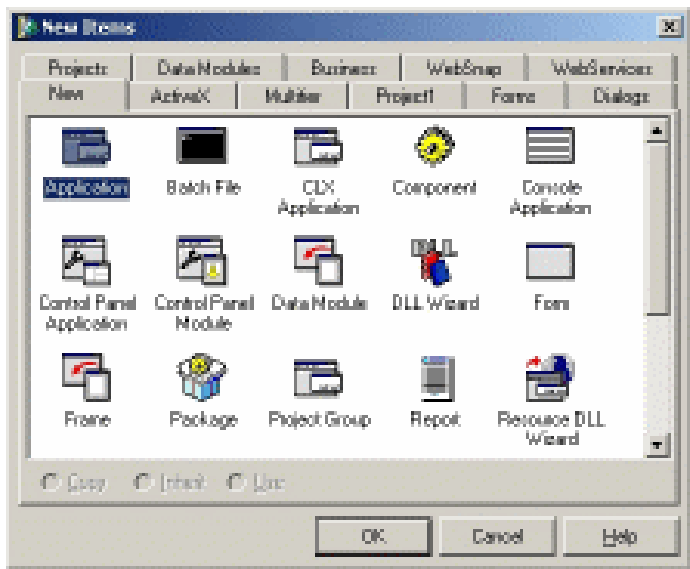


图 1-1 Delphi 全部向导程序（对象库）

如果只须创建普通的应用程序或者应用程序窗体、单元、数据模块等，可以通过菜单 File|New 导入，如图 1-2。

2. 提供了大量的组件

不仅 Delphi 本身附带的组件很多，而且在 Internet 上有非常多的组件包提供下载，可以安装到 Delphi 中直接使用，况且在 Delphi 中自己开发组件也是十分容易的。图 1-3 是 Delphi 的组件面板截图，其中主要是 Delphi 自带的组件，“lxpbuaa”是本书光盘（“第 6 章”目录下）附带的组件包，“Data ExControls”是我为公司开发的组件包。

通过向导程序生成一些对象后，如应用程序、窗体、数据模块等，从组件也拖放一些组件到窗体、数据模块中，最后设置必要的属性，再编写一些简单的代码（如处理事件），就可以按 F9 键运行程序了。

比如，我们选择菜单 File|New|Application，Delphi 自动生成包含一个窗体的应用程序，然后从 lxpbuaa 组件页（请首先安装光盘的 lxpbuaa.dpk 组件包）拖一个托盘组件（即图 1-3 中倒数第 4 个组

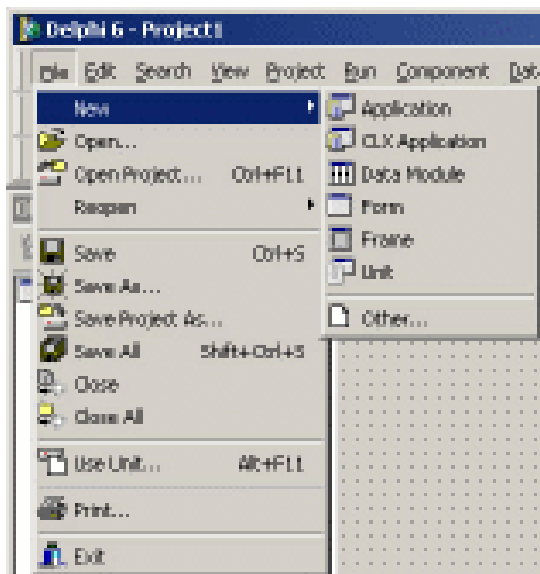


图 1-2 Delphi 常用向导程序

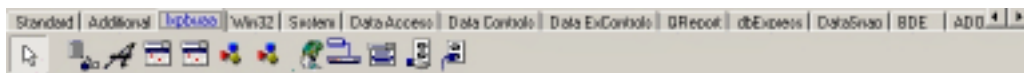


图 1-3 Delphi 组件页

件：TlxpTrayIcon) 到窗体，设置它的 Hint 属性为“托盘图标！”即可，不需要写一行代码，最后按 F9 键运行程序，我们就看到在 Windows 托盘区出现了该程序的托盘图标（见图 1-4 所示托盘区的第二个图标）。



图 1-4 托盘图标

很简单吧！如果你还不会安装组件包，请不要着急，本书后面章节有详细讲解。直接运行光盘上的“源代码\第 1 章”目录下的 Project1.exe 可以看到同样的效果。

1.2 本书的特点

本书名为《Delphi 精要》。那什么是 Delphi 的精要呢？在我看来，Delphi 精要可以分为两个方面：

(1) Object Pascal。

(2) VCL。

所以全书都是围绕着这两个方面来写作的。

1.2.1 本书有哪些内容，没有哪些内容

既然是讲精要，所以我没打算将本书写成一本 Delphi 大全。在有限的篇幅内，如果指望将所有问题和内容都包含进来，肯定是不现实的；如果执意这样做，就会主次难分，重点不突出，反而讲不成精要了！

所以，本书重点是在讲原理以及实践这些原理时必须注意的重要技巧，因而没有分章分节讨论从这些原理发展起来的专题，如 COM、数据库、网络程序、多媒体开发。但如果很好地认识了这些原理，那么理解和掌握这些专题知识的问题，也就迎刃而解了！

1.2.2 本书是如何写作的

原理非常重要，掌握这些原理是实现 Delphi（乃至使用其他工具）开发能力质变的必要前提，因此你不要以为本书所讲的内容对你的工作学习并没有多大帮助。很多朋友可能会认为开发技巧最为重要，因此“50 开发实例”、“100 实现技巧”之类的内容和书籍大受欢迎；CSDN 上的一些朋友讨论本书内容时，也强烈要求再多写一些实例、技巧。但是你应该认识到，一切技巧都是从原理演变而来的，懂得一个原理可以举一反三地推论出 10 个技巧，而即使掌握了 100 个技巧也未必能推论出第 101 个技巧。

你不用担心本书的内容会非常难以理解。本书力求深入浅出，尽可能用浅显易懂、简洁明快的语言表达作者的意思。书中给出了大量代码片断和完整实例，并作了详尽注释。为了满足初学者和基础较差朋友的需要，在第 3 章、第 5 章讲述 Object Pascal 和 VCL 精要之前，还分别专列一章（第 2 章、第 4 章）介绍 Object Pascal 和 VCL 的入门知识。

对于喜欢开发技巧的朋友 本书也会让你满意。在第 2 章、第 3 章、第 4 章、第 5 章讲述 Object Pascal 和 VCL 知识的过程中，不仅穿插了大量开发技巧的讨论和实现代码，而且第 6 章、第 9 章、第 10 章还是专门讲具体开发的，特别是第 9 章。

如果你希望将本书作为一本 Delphi 的知识手册，那么第 5 章中的“VCL 消息大全”和整个第 8 章也能满足你的要求。

本书的所有源代码和组件全部包含于光盘中，可以在学习和工作中直接使用，但是请不要用于商业目的。

1.2.3 如何阅读本书

每个人都有不同的读书方式。有的人喜欢深夜躺在床上阅读，有的人喜欢坐在计算机前边读边实践，也有的人喜欢坐在马桶上看书；有的开始时观其大略，需要时再仔细阅读，有的逐字研读，力求全盘吸收。

本书无法满足所有朋友的不同要求。如果你是在没有计算机的地方阅读本书，看到某些地方可能



希望自己动手实践，以验证我是否在故弄玄虚、大放厥词，这时候你可能有点光火——为什么不再花几行文字说详细点？而有些地方又可能让别的朋友觉得没必要讲得那么带劲——婆婆妈妈的，我都知道了，还说那么多！每当这个时候，希望你能消消气，因为我要照顾不同层次的读者。

因为是在讲精要、讲原理，所以一些地方未免显得有些艰深。不过没关系，先放一放往后面看，然后回过头来，很可能就豁然开朗了。Delphi 已经发展了 8 年（这还不包括非可视化的 Pascal 时代），8 年里，计算机技术日新月异。这么多年，Delphi 为全世界众多程序员推崇绝非偶然。经受了时间考验的东西肯定不是肤浅的，因此，希望在一朝一夕掌握整个 Delphi 的精要并不现实。

要完全掌握本书所讲的 Delphi 内容，对于不同基础的读者来说，所需要花费的时间和精力可能有较大的差别。对于高手来说，可能花一点时间翻翻就觉得没什么可看的了，有一定基础的需要几天至数周，而没有编程基础的 Delphi 初学者可能需要花上几月甚至更多的时间。



第 2 章 Object Pascal 入门

Object Pascal 是 Delphi 使用的开发语言，Delphi 的核心组件库 VCL 也是采用这种语言编写的。因此，可以说它是 Delphi 的基础。掌握 Object Pascal 精要是掌握 Delphi 精要的第一步。

本章是为第 3 章深入讲述 Object Pascal 精要作铺垫的部分，让大家对 Object Pascal 的特点有些初步的认识，更详细和深入的内容将在第 3 章讨论。

2.1 运算符

运算符是在程序中对各种数据类型常量和变量进行运算的符号。每种编程语言都必须定义必要的运算符，否则无法描述完整的表达式，就像每种语言中都必须定义“是、好像、并且、但是”等这些词一样，不然人们就不能相互对话。

本节将介绍 Pascal 中的各种运算符，并和 C/C++和 BASIC 的对应符号作个对比。

2.1.1 赋值运算符

如果你还是个初学者，可能对 Pascal 的赋值运算符非常的不满意，因为和其他多数语言直接使用“=”赋值不同，Pascal 中必须在“=”加上“:”，即“:=”。而“=”在 Pascal 中成了比较运算符，其他语言则多数采用“==”作为比较运算符，所以两两比较，算是扯平了。例如：

```
I:=5;
```

将 5 赋值给变量 I。

几种语言的赋值运算符比较见表 2-1。

表 2-1 赋值运算符

运算符	Pascal	C/C++	BASIC
赋值	:=	=	=

2.1.2 比较运算符

比较运算符是比较常量和变量大小关系的符号，比较结果是一个布尔值（True/False）。例如：

```
if I > 5 then DoSomething;
```

几种语言的比较运算符比较见表 2-2。



表 2-2 比较运算符

运算符	Pascal	C/C++	BASIC
等于	=	==	=或者 Is
不等于	<>	!=	<>
小于	<	<	<
大于	>	>	>
小于等于	<=	<=	<=
大于等于	>=	>=	>=

2.1.3 逻辑运算符

逻辑运算符是对逻辑表达式进行运算的符号。例如：

```
if (I > 5) and (I < 10) then DoSomething;
```

几种语言的逻辑运算符比较见表 2-3。

表 2-3 逻辑运算符

运算符	Pascal	C/C++	BASIC
逻辑与	and	&&	And
逻辑或	or		Or
逻辑非	not	!	Not

2.1.4 算术运算符

算术运算符是执行算术运算如加、减、乘、除等的符号。和其他大多数语言不同，在 Pascal 中，进行乘、除时，整数和浮点数使用的符号不同。例如：

```
I := J div C;
```

整数 J 和 C 作除法运算，并将结果赋值给 I。

几种语言的算术运算符比较见表 2-4。

表 2-4 算术运算符

运算符	Pascal	C/C++	BASIC
加	+	+	+
减	-	-	-
乘	*	*	*





续表

运算符	Pascal	C/C++	BASIC
除 (浮点数)	/	/	/
除 (整数)	div	/	/
取模	mod	%	Mod
指数	无	无	^

2.1.5 按位运算符

我们知道，变量在内存中是使用一些位 (Bit) 存储 0 或者 1 来保存的。按位运算符就是对位进行运算的符号。例如：

```
var
  I: Byte;      {Byte类型的取值范围在0~255，用8比特保存}
begin
  I := 0;       {此时I在内存中状态：00000000}
  I := not I;   {对各位取反，因此变为"11111111"，即255}
end;
```

几种语言的按位运算符比较见表 2-5。

表 2-5 按位运算符

运算符	Pascal	C/C++	BASIC
与	and	&	And
取反	not	~	Not
或	or		Or
异或	xor	^	Xor
左移	shl	<<	无
右移	shr	>>	无

2.1.6 其他运算符和模拟运算

对于一些复杂运算，Pascal 定义了特殊的运算符进行运算，也提供了一些函数和过程来模拟运算。大致有以下这些：





1. 对于集合类型

(1) in。判断集合是否包含一个元素，例如：

```
type
  TOneSet = set of (A, B, C);
var
  OneSet: TOneSet;
begin
  OneSet := [A, B];
  if A in OneSet then
    ShowMessage('集合 OneSet 包含 A');
end;
```

对于子界类型也是适用的，如：

```
var
  I: Integer;
begin
  I := 1;
  if I in [0..10] then      {[0..10]是一个子界表达式}
    ShowMessage('0<=I<=10');
end;
```

(2) 对一个集合的元素进行增加、减少时，除了使用+、-运算符外，还可以使用过程 Include 和 Exclude 如：

```
type
  TOneSet = set of (A, B, C);
var
  OneSet: TOneSet;
begin
  OneSet := [A];
  OneSet := OneSet + [B];  {等价于下一句}
  Include(OneSet, B);
  OneSet := OneSet - [B];  {等价于下一句}
  Exclude(OneSet, B);
end;
```

2. 对于对象类型转化

(1) 可实现使用 is 进行类型兼容性判断，然后用 as 转化。例如：



```
var
  Obj: TObject;
  Button: TButton;
begin
  .....
  if Obj is TButton then      { 如果Obj是TButton类型或者其子类}
    Button := Obj as TButton; { 将Obj转化为TButton类型并赋值给Button}
end;
```

(2) 也可以使用 TObject.InheritsFrom 代替 is 运算符，例如：

```
var
  Obj: TObject;
  Button: TButton;
begin
  .....
  if Obj.InheritsFrom(TButton) then { 如果Obj是TButton类型或者其子类}
    Button := Obj as TButton;      { 将Obj转化为TButton类型并赋值给Button}
end;
```

3. 对指针操作

(1) 符号@和^。@用于取得一个变量的地址指针；^用来取得一个指针对应的数据，也可以用于声明指针类型。例如：

```
type
  PInteger = ^Integer; { 在类型名前面加上^可以声明其对应的指针类型}
var
  I, J: Integer;
  PI: PInteger;
begin
  I := 5;
  PI := @I;           { 取得变量I的地址指针}
  J := PI^;           { 从PI中取出值，此时I=J}
end;
```

有些语言能对指针作加、减等运算，但是 Pascal 中能进行这类运算的只有 PChar 类型，所以这里就不介绍了。只须记住，在 Pascal 中一般不允许对指针直接作加、减运算就可以了。

(2) 使用函数 Addr 可以代替@符号取得变量地址。如：



```
PI := Addr(I);
```

判断一个指针是否有指向时，除了使用：

```
if P <> nil then DoSomething;
```

这样的形式外，还可以调用函数 Assigned，它们是等价的：

```
if Assigned(P) then DoSomething;
```

4. 加减运算

Inc 和 Dec：

```
procedure Inc(var X [ ; N: Longint ]);
procedure Dec(var X[ ; N: Longint]);
```

它们可运用在所有 ordinal 类型（参看 3.1.1 数据类型概述）的变量上。如：

```
var
  I: Integer;
begin
  I := 1;
  Inc(I);           {此时I = 2;Inc和Dec默认的增减量是1,相当于I := I + 1}
  Inc(I, 2);        {此时I = 3;相当于I := I + 2}
end;
```

值得一提的是，它们也可以用于指针类型，表示对指针移位 SizeOf(X)*N 个字节。如：

```
var
  P: PChar;
begin
  P := 'China';
  Inc(P);           {此时指向从'C'移动到'h'上}
  ShowMessage(P);   {显示'hina'}
  Dec(P);           {此时指向从'h'移动到'C'上}
  ShowMessage(P);   {显示'China'}
end;
```

2.2 常量和变量

从外观上看，常量和变量都是一些符号；从内部看，它们代表某块内存中保存的数据。常量的值是在声明时就被给定的，程序运行时不能改变，但是变量在声明时一般不能给定初始值（全局变量除外），在运行时可以改变。

和 C/C++ 不同，Pascal 中声明常量和变量时必须在 `const` 或者 `var` 块中进行，而不能在 `begin...end` 块中声明。

2.2.1 无类型常量和有类型常量

看如下的常量声明：

```
const
  I = 5;
  WM_MYMSG = WM_USER + 100;
  ErrorInfo = '没有足够的参数';
```

这样就定义了三个常量。

如上格式定义的常量，称为无类型常量，因为声明时并没有明确指定数据的类型，而是依靠编译器自动指定。如 `I` 和 `WM_MYMSG` 被指定为整型，`ErrorInfo` 被指定为字符串。编译器自动指定时，一般是尽可能采用占用内存最小的类型，如上面的 `I` 被指定为 `ShortInt` 类型（占用 8 个字节，范围在 -128~127）。

在 Pascal 中，也可以在声明变量时指定数据类型，例如：

```
const
  I: Integer = 5;
```

这样，`I` 被编译为 `Integer` 类型而不是 `ShortInt`。这样的常量称为有类型常量。

2.2.2 全局变量和局部变量

在一个单元中声明的变量（不包括类中声明的）可分为全局变量和局部变量。在过程和函数内部声明的为局部变量，只能在过程和函数内部使用；其他的为全局变量，全局变量可在整个单元中使用。如：

```
.....
var
  Form1: TForm1;           { 这是全局变量 }
  OneInt: Integer = 5;     { 这也是全局变量 }

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;              { 这是局部变量 }
```



```
begin
    .....
end;
```

大家注意例子中的变量 OneInt，我们像对待常量一样给它指定了值 5。它的意思是给 OneInt 指定初始值 5。

注意：全局变量可以被指定初始值，但是局部变量则不可以。

2.2.3 声明时使用编译时函数

在 Object Pascal 中声明常量、变量和类型时，可以使用编译时函数，它们包括：

- (1) Ord。取得有序变量的值在取值范围中所在的顺序（即位置），具体可参看 3.1.1 节。
- (2) Chr。取得 ASCII 码对应的字符。
- (3) Trunc、Round。将浮点数取整。
- (4) High、Low。分别取得有序类型/变量的末序和始序的值，具体可参看 3.1.1 节。
- (5) SizeOf。取得类型或者变量所需的内存大小（即字节数）。

例如：

```
type
    TOneArray = Array[0..5] of Integer;    {声明一个整数数据类型}

var
    R1: Word = Trunc(12.6);                {R1被初始化为12}
    R2: Word = Round(12.6);                {R2被初始化为13}
    H: Byte = High(TOneArray);             {H被初始化为5}
    L: Byte = Low(TOneArray);              {L被初始化为0}

const
    I: Integer = Ord('A');                 {I值为65}
    C: Char = Chr(65);                     {C值为'A'}
    J: Integer = SizeOf(I);                 {J值为4}
    K: Integer = SizeOf(Integer);          {K值也为4}
```

所谓编译时函数，是指实现于编译器中间的可调用的代码段。在 IDE 中写下任何一个编译时函数名，然后按住 Ctrl 键并在它上面单击鼠标左键，你会发现 Delphi 将你引导到 System 单元，但是并不能看到该函数的任何信息（包括声明和实现）。这是因为它们被构筑在编译器中，并没有源代码。

因为被构筑在编译器中，所以在声明和实现时都可以调用它们。

2.2.4 资源字符串

资源字符串是一种特殊的常量。例如：

```
resourcestring
  Author = '罗小平';
  Age = '25';
  Sex = '男';
```

这样就定义了三个资源字符串，在程序中可以将它们当作字符串常量一样使用。不同的是，资源字符串被编译到程序的资源中，而不是像普通字符串常量和变量被内嵌入源代码，因此，实现了字符串和源代码的分离，这对编写跨语言平台非常有好处，当运行平台的语言环境发生变化时，只须修改这些资源字符串即可，而不须修改程序。

2.3 过程和函数

过程和函数都是用来完成特定功能的一个代码块，可以在别的地方被调用。

2.3.1 过程和函数的区别

除了函数可以有返回值而过程没有返回值外，你可以认为它们是完全相同的。因此，在 Delphi 的 IDE 和在线帮助中，通常用过程来统称过程和函数（也有统称为例程的，但是在本书中采用过程的说法）。当一个过程或者函数属于一个类时，可以统称为方法。

例如：

```
function GetApplicationPath(ShowResult: Boolean): String;
{ 这是一个函数，返回应用程序文件所在目录 }
begin
  Result := ParamStr(0);
  Result := ExtractFilePath(Result);
  if ShowResult then
    ShowMessage('应用程序路径是：' + Result);
end;

procedure ShowInfo(Info: String);
{ 这是一个过程，显示信息 Info }
begin
  ShowMessage('应用程序路径是：' + Info);
end;
```



```

procedure TForm1.Button1Click(Sender: TObject);
{Button1Click和下面的Button2Click都是方法，属于类TForm1}
var
    S: String;
begin
    S := GetApplicationPath(False); {调用函数GetApplicationPath}
    ShowInfo(S);                  {调用过程ShowInfo}
end;

procedure TForm1.Button2Click(Sender: TObject);
{方法Button2Click和方法Button1Click实现的功能是完全一样的}
begin
    GetApplicationPath(True);
end;

```

看上面的例子时，注意三点：

- (1) 在函数内部有一个预定义的变量 `Result`，其类型和函数定义的返回值类型相同。最后赋给它的值，就是函数的返回值。尽管也可以用函数名来代替 `Result` 使用，但是通常不这样做，因为 `Result` 要简单明了得多，且在修改函数名后，无须对返回语句进行修改。
- (2) 如果不需要返回值，函数也可以当作过程一样调用，如例子中的 `Button2Click`。
- (3) 当函数和过程没有参数时，可以直接使用名字调用，而不需要加“()”这一对空括号。

2.3.2 子过程

定义在函数和过程内部的函数和过程称为子过程。子过程只在母过程内部有效。例如：

```

procedure TForm1.Button2Click(Sender: TObject);
var
    S: String;
{实现Button2Click的子过程ShowInfo。ShowInfo只能在Button2Click中被调用}
procedure ShowInfo(Info: String);
begin
    ShowMessage(Info);
end;
begin
    S := 'lxpbuaa';
    ShowInfo(S);
end;

```





2.4 数据类型

本节只打算简要介绍 Object Pascal 中的数据类型，在第 3 章中将讨论它们的深入知识。

2.4.1 常用类型和复杂类型

粗略地讲，可以将 Object Pascal 中的数据类型分为两类：

- (1) 常用类型。
- (2) 复杂类型。

“常用类型”并不是一个严格的说法，它包含常用、简单两层意思。通常，它是用来定义简单的变量，由于是简单的，因此，往往也是比较常用的。它们包括：

整数(Integer) 实数(Real) 布尔类型(Boolean) 字符串(Character、String) 枚举(Enumerated) 子界(Subrange) 和可变类型(Variant) 等。

但是现实世界是复杂的，仅仅使用这些类型是无法满足需要的，所以在此基础上，Object Pascal 也提供了一些更复杂的数据类型，通常称它们为构造(Structured) 类型。如：

集合(Set) 数组(Array) 记录(Record) 文件(File) 类(Class) 类引用(Class Reference) 接口(Interface) 等。

2.4.2 不同语言的数据类型对照表

我在这里给出一个 Pascal、C/C++、BASIC 数据类型的对照表，方便编写和调用 DLL (动态链接库) 和 OBJ (目标文件) 时查阅。参见表 2-6 (带*前缀的为 Pascal 中向后兼容的类型，新开发不应该再使用；带^前缀的为 C++ Builder 特有，用于模拟 Pascal 对应类型)。

表 2-6 不同语言的数据类型对照表

数据类型	Pascal	C/C++	BASIC
8 位有符号整数	ShortInt	char	无
8 位无符号整数	Byte	BYTE , unsigned short	Byte
16 位有符号整数	SmallInt	short	Short
16 位无符号整数	Word	unsigned short	无
32 位有符号整数	Integer , LongInt	int , long	Integer Long
32 位无符号整数	Cardinal , LongWord/DWORD	unsigned long	无
64 位有符号整数	Int64	_int64	无
4 字节浮点数	Single	float	Single
6 字节浮点数	*Real48	无	无



续表

数据类型	Pascal	C/C++	BASIC
8 字节浮点数	Double	double	Double
10 字节浮点数	*Extended	long double	无
64 位货币类型	Currency	无	Currency
8 字节日期/时间	TDate/TDateTime	无	Date
16 字节可变类型	Variant , OleVariant	VARIANT , ^Variant , ^OleVariant	Variant
1 字节字符	Char , AnsiChar	char	无
2 字节字符	WideChar	WCHAR	无
短字符串	*ShortString	无	无
长字符串	AnsiString/String	^AnsiString	String
宽字符串	WideString	^WideString	无
NULL 结束的字符串	PChar , PAnsiChar	char*	无
NULL 结束的宽字符串	PWideChar	LPCWSTR	无
1 字节布尔类型	Boolean , ByteBool	(任何 1 字节数)	无
2 字节布尔类型	WordBool	(任何 2 字节数)	Boolean
4 字节布尔类型	BOOL , LongBool	BOOL	无

2.5 程序流程控制

几乎任何语言都包含三种程序流程控制方法：

- (1) 顺序；
- (2) 条件分支；
- (3) 循环。

顺序，是指程序按照程序语句的顺序执行，例如语句 A 在 B 前面，所以首先执行 A 再执行 B；条件分支，是指根据不同的条件执行不同的语句；循环，是指由特定的条件决定某些语句重复执行的次数。

有些语言还有跳转的流程控制方法，它是指在程序的某个地方跳到另一个地方去执行语句。如 FORTRAN 和 Pascal 可使用 goto 进行跳转。

2.5.1 条件分支

在 Pascal 中，可以使用 if...else 和 case of...else...end 两种方式实现条件分支控制。例如 if 方式：

```

var
  I: Integer;
begin
  if I > 0 then      { 如果I大于0 }
    DoSomething1
  else if I < 0 then { 如果I小于0, else if 块不是必须的 }
    DoSomething2
  else              { 如果是其他条件, 通常用来实现默认处理。else块不是必须的 }
    DoSomething
end;

```

使用 if 方式要**注意**的是：else（包括 else if）关键字之前的语句不能包括“;”。

又如 case 方式：

```

var
  I: Integer;
begin
  case I of
    1:      { 如果I=1 }
      DoSomething1;
    2,3:    { 如果I=2或者3, 相当于if I in [2,3] }
      DoSomething2;
    4..10:  { 如果I=在4到10范围内(含4和10), 相当于if I in [4..10] }
      DoSomething3;
  else     { 如果是其他条件, 通常用来实现默认处理。else块不是必须的 }
    DoSomething;
  end;
end;

```

使用 case 方式要**注意**的是：它只能用有序类型（参见 3.1.1 节）变量作为条件因子，其他的如字符串则是不行的。

2.5.2 循环

在 Pascal 中，可以使用 for to/down to do、while...do 和 repeat...until 三种方式实现循环控制。例如 for 方式：

```

var
  I: Integer;
begin

```



```
for I := 0 to 10 do {I从0变化到10, DoSomething被执行11次}
{或者for I := 10 downto 0 do, I从10变化到0, DoSomething被执行11次}
  DoSomething;
end;
```

使用 for 方式要注意的是：循环变量的步长是固定为 1 的，无法改变；不能在循环块中用代码改变循环变量的值。

又如 while 方式：

```
var
  I: Integer;
begin
  I := 0;
  while I <= 10 do {和for I := 0 to 10 do的功能是相同的}
  begin
    DoSomething;
    Inc(I);
  end;
end;
```

while 方式的特点是可以自己控制循环因子的变化，和 for 相比，灵活性更大。

repeat 是从 while 演化出来的。while 首先判断条件是否成立再执行，而 repeat 是首先执行，再判断条件是否成立。如：

```
var
  I: Integer;
begin
  I := 0;
  repeat {和while I <= 10 do功能相同}
  DoSomething;
  Inc(I);
  until I = 10;
end;
```

2.5.3 跳转

跳转是属于结构化编程中的概念，尽管现在大部分程序员已经进入面向对象编程（OOP）时代，但是 Object Pascal 仍然保留了跳转功能。如下例子可以实现跳转：

```

var
  I: Integer;
label
  labelDo;           { 跳转必须有一个名字，必须事先用label关键字声明}
begin
  I := -1;
  labelDo:
    Inc(I);           { 这一句属于跳转labelDo的语句块}
  if I < 3 then
  begin
    ShowMessage('I的当前值为：' + IntToStr(I));
    goto labelDo;     { 跳转到labelDo执行语句}
  end;
end;

```


在面向对象编程中，通常不再使用跳转语句来控制程序流程，因为这很容易引起流程混乱。跳转的功能，很容易就可以用函数或者过程配合其他流程控制方式来实现。如上例可以用一个子过程来实现“跳转”功能：

```

var
  I: Integer;
  procedure AddOne;   { 类似于跳转labelDo}
  begin
    Inc(I);
  end;
begin
  for I := 0 to 3 do
  begin
    ShowMessage('I的当前值为：' + IntToStr(I));
    AddOne;
  end;
end;

```

2.5


 程序流程控制

2.5.4 用过程辅助实现流程控制

Object Pascal 定义以下几个过程来辅助流程控制：

1. **procedure** Abort;

激发一个静态异常（即异常信息不显示）并退出当前过程或者 try...except 块。例如：

```

procedure TForm1.Button2Click(Sender: TObject);
begin

```



```

if True then
  Abort;      { 激发一个静态异常后直接退出Button2Click，不会执行后面的
               ShowMessage语句}
  ShowMessage('处理完毕');
end;

```

如果 Abort 被包含在一个 try...except 块中，则只退出 try...except，然后执行 except...end 块，最后再继续执行下面的语句。如果包含在 try...finally 块中，则执行 finally...end 块然后退出过程。如：

```

procedure TForm1.Button2Click(Sender: TObject);
begin
  try
    if True then
      Abort ;
    except      {Abort激发的异常被捕捉，因此执行except...end中的异常处理代码}
      ShowMessage('Abort');
    end;
    ShowMessage('处理完毕');      {执行，因为异常已经被处理。如果使用
                                   try...finally，则这句不会被执行}
  end;

```

2. **procedure** Exit;

正常退出过程。如果处在 try...finally 块中，则执行 finally...end 块然后退出过程。例如：

```

begin
  try
    if True then
      Exit;
    finally
      ShowMessage('Abort');      {执行}
    end;
    ShowMessage('处理完毕');      {不执行}
  end;

```

3. **procedure** Halt([Exitcode: Integer]);

非正常结束一个应用程序，Exitcode 为可选的退出码参数，此参数由操作系统接收。如果是非控制台（即有图形化界面）程序，一般调用 Application.Terminate 来实现正常结束。

4. **procedure** RunError ([Errorcode: Byte]);

生成一个运行时错误（由可选参数 Errorcode 指定，默认为 0）并结束程序执行。

5. **procedure** Continue;

使用在循环语句中，结束当前循环并进入下一个循环。如果位于 try...finally 块中，则会首先执行 finally...end 再进入下一个循环。例如：

```
var
  I: Integer;
begin
  for I := 0 to 2 do
  begin
    try
      if I < 2 then
        Continue;
      ShowMessage(IntToStr(I));
    finally
      ShowMessage('finally');
    end;
  end;
end;
```

上面例子的执行结果是：首先显示两次'finally'（分别对应 I=0 和 1），然后显示一次 I（对应 I=2），最后显示一次 I（对应 I=2）。

6. **procedure** Break;

使用在循环语句中，完全结束本循环。如果位于 try...finally 块中，则 finally...end 块也会得到执行。例如：

```
var
  I: Integer;
begin
  for I := 0 to 2 do
  begin
    try
      if I > 0 then
        Break;
      ShowMessage(IntToStr(I));
    finally
      ShowMessage('finally');
    end;
  end;
end;
```



上面例子的执行结果是：首先分别显示一次 I 和 'finally' (对应 I=0)，然后显示一次 'finally' (对应 I=1)。

注意：如果使用了嵌套循环（即循环中包含子循环），那么 Continue 和 Break 只对它所属的子循环起作用。

2.6 单元的组织结构

在 Delphi 中，一个正在开发的应用程序可以被称作项目或者工程。一般地，一个项目主要由 dpr（项目）、pas（单元）和 dfm（窗体）三种文件组成，另外还有一些附属文件，如 res（资源）文件等。其中项目文件可以被看做是一种特殊的单元。在源代码中，项目文件用关键字 program 标识，单元文件用 unit 标识。

通常，一个项目只有惟一的 dpr 文件。一个 dfm 文件总是有对应的 pas 文件，但是 pas 文件可以没有对应的 pas 文件。

如果打开 Delphi，选择菜单 File|New|Application，则可以新建一个项目。该项目包括一个项目文件——Project1.dpr、一个窗体文件——Unit1.dfm 和一个对应的单元文件——Unit1.pas。选择菜单 Project|View Source 可以看到项目文件的内容；在窗体上单击右键，从弹出的菜单中选择 View as Text 可以看到窗体文件的内容。

这些文件的内容都必须按照一定的组织结构来编写，编译器按照既定的组织结构来识别这些内容并进行编译。在本节里，我们讨论 program 和 unit 两种单元文件的组织结构。

2.6.1 Program 的组织结构

一个项目文件被关键字 program 标识，因此，在这里我将项目文件的源代码称作 program。以下是一个简单的 program：

```
program Project1; { 文件定义：一个名为Project1的项目文件 }

uses
    Forms,
    Unit1 in 'Unit1.pas' {Form1},
    Dialogs;

{ 在这里可以定义一些变量和常量 }
var
    AppMsg: String;

{$R *.res} { $R是一个编译指令，此处表示要编译资源文件Project1.res }
```

```

{ 这个部分可以实现一个函数和过程 }
procedure AppStart(AppMsg: String);
begin
    ShowMessage(AppMsg);
end;

{begin...end部分是program的主体, 这里面的代码是可以运行的}
begin
    {应用程序初始化}
    Application.Initialize;
    Application.Title := 'lxpbuaa';
    {创建主窗体}
    Application.CreateForm(TForm1, Form1);
    AppMsg := '应用程序马上开始运行。';
    AppStart(AppMsg);
    {应用程序开始运行}
    Application.Run;
end.

```

2.6.2 Unit 的组织结构

一个单元文件被关键字 `unit` 标识, 因此, 在这里我将单元文件的源代码称作 `unit`。以下是一个完整的 `unit` :

```

unit Unit1; {文件定义: 一个名为Unit1的单元文件}

interface    {在这个部分声明可供其他单元使用的变量、常量、类型、函数和过程}

uses         {interface部分的uses内容对整个单元都有效}
    Windows, Messages, Classes, Controls, Forms, Dialogs, Contnrs;

type         {声明类型}
    TForm1 = class(TForm)
    private
        procedure ShowInfo(Info: String);
    public
        { Public declarations }
    end;

    {声明函数和过程}
    procedure ShowInfo(Info: String);

```




```

var                                { 声明变量 }
    Form1: TForm1;

implementation { 在这个部分完成单元的私有声明，并实现interface声明的类、函数和
                  过程 }

uses                                { implementation部分的uses内容只对implementation有效。
                  interface不需要而implementation需要的单元应该在这里引用 }
    SysUtils, Variants;

var                                { implementation部分可以和interface部分一样进行声明 }
    ObjList: TObjectList;

{$R *.dfm}                          { 编译对应的dfm文件 }

{ 实现interface部分声明的函数和过程 }
procedure ShowInfo(Info: String);
begin
    ShowMessage(Info);
end;

{ 实现interface部分声明的类 }
{ TForm1 }

procedure TForm1.ShowInfo(Info: String);
begin
    ShowMessage(Info);
end;

{ 单元初始化部分 }
initialization
    ObjList := TObjectList.Create;

{ 单元终止部分 }
finalization
    FreeAndNil(ObjList);

end.

```

从上面的 Unit1 单元可以看到，一个 unit 可以包含五个部分：

(1) unit 关键字部分, 指定单元的名字。

(2) interface 部分。从关键字 interface 到 implementation 为止的内容, 都是属于这个部分。该部分可以声明变量、常量、类型、函数和过程, 而且它们对于其他单元都是可见的。

(3) implementation 部分。在这个部分也可以完成 interface 具有的声明功能, 但是它们对于其他单元是不可见的, 属本单元私有; 同时完成类、函数和过程的实现。

以上三个部分是一个 unit 必须的。接下来的两个部分是可选的。

(4) initialization。在这个部分可以完成单元的初始化工作。如果将一个单元比作一个类, 我们知道类的初始化是在构造函数 Create 中完成的, 所以 initialization 部分就相当于单元的构造函数。

(5) finalization。在这个部分可以完成单元的终止, 完成类似于类的析构函数 Destroy 的功能。

需要注意的是: 如果几个单元都有 initialization/finalization 部分, 则它们的执行顺序与这些单元在 program 的 uses 字句中的出现顺序一致。所以应该避免 initialization/finalization 部分的代码执行时依赖于它们的执行顺序。

2.6.3 单元循环引用

单元不能被循环引用 (Circular unit reference) 的。循环引用的意思是: A 引用了 B, 而 B 又引用 A, 且都是在 interface 部分进行引用。如下面的单元通不过编译:

```
unit Unit1;

interface

uses
  Unit2;
.....

unit Unit2;

interface

uses
  Unit1;
.....
```

但是如果引用不全发生在 interface 部分, 即至少有一个在 implementation 部分, 则是允许的。因此, 当你需要两个单元相互引用时, 应该将其中的一个引用放置在 implementation 部分, 否则不能通过编译。

为了避免将来可能的循环引用, 对于只在实现部分使用的单元, 通常我们都将它写在



implementation 而不是 interface 部分。例如我们编写一个取得一个整型动态数组中最大的元素，需要用到 Math 单元的 Max 函数，此时代码书写应该按照如下所示：

```
unit Unit2;

interface

{ 因为在声明部分只有一个函数GetMax，只须引用System单元，而该单元是自动引用的，所以在源代码中，接口部分没有任何引用单元的代码行}

function GetMax(IntDynArray: Array of Integer): Integer;

implementation

uses Math;      { 在这里而不是接口部分引用单元Math}

function GetMax(IntDynArray: Array of Integer): Integer;
var
    L,I: Integer;
begin
    Result := 0;
    L := Length(IntDynArray);
    if L = 0 then Exit
    else
    begin
        Result := IntDynArray[Low(IntDynArray)];
        for I := Low(IntDynArray)+1 to High(IntDynArray) do
            Result := Max(Result, IntDynArray[I]);
        end;
    end;
end;

end.
```

2.7 with...do 语句的用法

with...do 语句用来指定一个块中的字段（记录的或者对象的）属性和方法所属的记录或者对象。Delphi 初学者可能已经习惯了如下的代码书写格式：

```
var
    Button: TButton;
```

```
begin
  Button := TButton.Create(Self);
  Button.Parent := Self;
  Button.Left := 50;
  Button.Top := 50;
  .....
end;
```

每行都写一个 Button 是否让你感觉厌烦。Object Pascal 提供它特有的 with...do 语句，可以帮你消除这个烦恼。上面的代码可以写为：

```
var
  Button: TButton;
begin
  Button := TButton.Create(Self);
  with Button do { 编译器会知道下面的三个属性属于Button }
  begin
    Parent := Self;
    Left := 50;
    Top := 50;
    .....
  end;
end;
```

with...do 中可以包含多个记录或者对象，用逗号隔开。with...do 也可以嵌套使用。

2.8 IDE 的快捷键列表

使用快捷键可以加快设计速度。下面列出了使用 Delphi 开发程序时常用的快捷键，可根据实际情况，有选择地熟练掌握。如果你须要知道更多的快捷键设置，请在 Delphi 在线帮助的索引中输入“Classic keystroke mapping”查找。

1. 组件设计操作类

Ctrl + 方向键：将所选组件的位置移动一个像素。

Shift + 方向键：将所选组件的大小改变一个像素。

Ctrl + Shift + 方向键：将所选组件的位置移动一个栅格。

TAB：选择当前组件的下一个组件。

Shift + TAB：选择当前组件的上一个组件。

方向键：选中此方向上离当前组件最近的组件。

Shift + 鼠标左键单击：选择多个组件。





Del：删除所选组件。

Esc：选择当前组件的容器（通常是 TPanel、TGroupBox、TForm 等）。

Ctrl + 鼠标左键按下拖动：可选择一个容器内的多个组件。

注意：当选择了多个组件时，可以一次改变它们共同有的属性。

2. 程序编写类

F1：显示光标所在单词的帮助信息。

Shift + Alt + 方向键：选择块。

Ctrl + K + I，Ctrl + K + U：将所选择的代码整体右移或者左移。

Ctrl + Shift + 上/下方向键：在过程的声明和实现间切换。

Ctrl + Shift + 数字键（0~9）：设置书签。

Ctrl + 数字键（0~9）：返回到书签位置。

Ctrl + Shift + R：开始/完成录制宏。

Ctrl + Shift + P：应用宏。

Ctrl + J：插入标准语法代码。

Ctrl + Y：删除光标所在行。

Ctrl + Q + Y：删除光标位置后的该行内容。

Ctrl + BackSpace：向前删除一个单词。

Ctrl + K + E/F：将单词转化为全小/大写。

Ctrl + O + U：将字母作大小写转化。

Ctrl + Home：到达单元头部。

Ctrl + End：到达单元尾部。

Ctrl + F：调出搜索对话框。

F3：继续搜索。

Ctrl + R：调出替换对话框。

Ctrl + Shift + G：插入 GUID 号。

Ctrl + Shift + C：完成已声明类的实现。

3. 程序管理类

F11：在 Form/Unit 和 Object Inspector（对象检查器）之间切换。

F12：在 Form 和 Unit 之间切换。

Ctrl + Alt + F11：弹出 Project Options（工程管理器）。

第 3 章 Object Pascal 精要

崔浩，字伯渊，清河东武城（今山东武城西）人。北魏太武帝初拜博士祭酒，赐爵武城子。历太常卿、侍中、特进抚军大将军、左光禄大夫、司徒。他根据其母卢氏“口授”作成《食经》一书，是我国古代最早集中记载饮食文化的典籍。古代厨师无不研习《食经》。

今天，我们有了 Object Pascal。Borland 公司据以完成 Delphi。

Delphi 的真正精髓乃是 Object Pascal。Object Pascal 是 VCL 架构的基础和 Delphi 开发的原理所在。因此，如果把 Delphi 爱好者比作厨师，那么 Object Pascal 就可以看做《食经》。

不少朋友也许会认为我对 Object Pascal 的重要性夸大其词了。因为在很多人的印象中，在各种书籍、资料、文章、帖子上，“Object Pascal”比“TButton”的出场率低得多，而且“TButton”也还似乎不过是“路人甲”、“士兵乙”之流的货色呢！

老实讲，我也是在使用 Delphi 一段时间后系统学习 Object Pascal 的。之所以下决心系统学习 Object Pascal，是因为在使用 Delphi 一段时间后，我发现：

（1）虽然可以用 Delphi 做各种各样的界面，但是它们没有灵活性，好像手脚被捆，总是被死气笼罩着。因为都是通过设计时放置组、控件来实现的，对程序运行时怎么用程序去控制它们，我就感到茫然。

（2）写的代码老是编译时提示“ incompatible types”、运行时出现“ raised exception、access violation”。

（3）不知道如何使用指针、记录。

.....

在郁闷和困惑中折腾了一段时间，我感觉到自己最根本的问题是贫血，而不是练习的招数太少。贫血的习武人练习的招数再多，出击也是无力的，因此必须首先补血，加强对 Delphi 的基础——Object Pascal 的学习。

根据我的经验和理解，Object Pascal 的重点内容（或者说 Object Pascal 精要）有以下几方面：

（1）数据类型及其兼容、转化关系。

（2）过程和函数。

（3）类和类成员。

另外，编译指令也比较重要。因此本章将从这四方面来讲述 Object Pascal 精要，但重点是前三方面。

3.1 数据类型及其相互关系

Object Pascal 的数据类型就好比菜市上五花八门的菜。走进菜市，你是不是常常感到茫然，不知

道买哪样好？Integer 和 Word 差不多，哪个价廉物美呢？今天我是否该吃指针这道菜呢？哪些菜可以相互替代？

本节将对 Object Pascal 的数据类型作全面的介绍，并揭示各种数据类型的内存管理方法，最后讨论它们相互的兼容关系和转化方法。

有了本节的知识，就能对各种数据类型有比较全面、深入的认识，并掌握它们的相互关系，最终能够在编程时能合理、准确地选择使用它们。

3.1.1 数据类型概述

Object Pascal 中定义了大量数据类型。不同的数据类型具有不同的用途，比如 Integer 用来操作整数，Real 用来操作实数。但是它们之间也不是严格划清界限、老死不相往来的，也就是说，某些数据类型之间具有兼容性和转化性。

在深入讨论之前，我们首先要对数据类型大家族有个总体认识。

大家知道，一大帮子人凑到一起，就必须构建一个相应的组织体系，在他们之间建立起隶属、管理、信任和合作的关系，从而用这个体系和这些关系将所有的人组织起来，否则就成了一帮乌合之众。乌合之众不但不能成事，而且行动起来还会坏事。

Object Pascal 中的数据类型也是一样，必须有个明确的体系，否则根本无法在上面建筑起功能强大的 Delphi。

因此，我首先要拿出一张珍藏很久的数据类型全家福，所有数据类型都按照各自的辈分站好了位置。其中黑体部分表示是可以直接使用的类型标识符，也就是说，可以用来直接声明变量；带*的表示向后兼容的类型，在新的开发中不要再使用它们；带下划线的是可以直接使用的类型标识符，同时，它们还是所属类型家庭的基本类型，基本类型是我们声明变量时常用的类型。比如 Integer 和 Cardinal 是整数类型的基本类型，我们经常使用这两个类型来声明整数变量。其他如“Simple”、“Ordinal”可以看做大家族下各家的家长。

Simple

Ordinal

类型	取值范围	空间
Integer		
Integer	-2147483648~2147483647	signed 32-bit
Cardinal	0~4294967295	unsigned 32-bit
Shortint	-128~127	signed 8-bit
Smallint	-32768~32767	signed 16-bit
Longint	-2147483648~2147483647	signed 32-bit
Int64	$-2^{63} \sim 2^{63} - 1$	signed 64-bit
Byte	0~255	unsigned 8-bit
Word	0~65535	unsigned 16-bit
Longword	0~4294967295	unsigned 32-bit

Character					
	<u>Char</u> AnsiChar WideChar				
Boolean					
	<u>Boolean</u>	ByteBool	WordBool	LongBool	
Enumerated					
Subrange					
Real					
	类型	取值范围	精度	空间	
	<u>Real</u>	$5.0 \times 10^{-324} \sim 1.7 \times 10^{308}$	15-16	8B	
	*Real48	$2.9 \times 10^{-39} \sim 1.7 \times 10^{38}$	11-12	6B	
	Single	$1.5 \times 10^{-45} \sim 3.4 \times 10^{38}$	7-8	4B	
	Double	$5.0 \times 10^{-324} \sim 1.7 \times 10^{308}$	15-16	8B	
	Extended	$3.6 \times 10^{-4951} \sim 1.1 \times 10^{4932}$	19-20	10B	
	*Comp	$-2^{63}+1 \sim 2^{63}-1$	19-20	8B	
	Currency	$-922337203685477.5808 \sim 922337203685477.5807$	19-20	8B	
String					
	*ShortString	AnsiString	WideString	PChar	PAnsiChar PWideChar
Structured					
	Set	Array	Record	File	Class Class reference Interface
Pointer					
	无类型指针	有类型指针			
Procedural					
	普通过程类型	对象过程类型			
Variant					
	Variant	OleVariant			

如果你对它们中的某些成员还比较陌生，不要紧，请看后面的详细说明。Object Pascal 将 Ordinal 和 Real 归于 Simple（简单）旗下，不知你是否已经彻底弄清楚这两个简单类型了呢！

1. Ordinal（有序）类型

简单地说，Simple（简单）类型规定了一个数据取值范围。而 Ordinal 类型在此基础上，还增加了有序的特性。

有序，是说变量的取值是有序的，那么对于这样的变量，我们可以称为有序变量。有序变量只能在取值范围中取某个位置上的值，或者说，已经初始化后的有序变量的值总是落在取值范围的某个确定位置。

有序变量取值必须落在既定的范围内，不能越界；如果强行越界，将得到错误的运算结果。比如下面的代码：



```
var
  B: Byte;
begin
  B := 256;
```

在编译时将无法通过，提示：

[Error] Unit1.pas(30): Constant expression violates subrange bounds

上面的错误信息的意思是：常数表达式（即 256）越界，因为 Byte 变量的取值范围在 0~255。

如果是在运行时越界，则可能有两种情况，这两种情况和编译指令 \$R 有关：

(1) {\$R+} 状态，将执行越界检查。此时越界会抛出 ERangeError 类型的异常。我们看下面的代码：

```
var
  B: Byte;
begin
  B := 255;
  {$R+}
  B := B + 1;
  {$R-}
  ShowMessage(IntToStr(B));
end;
```

运行到 “B := B + 1;” 后会出现如图 3-1 的错误。



图 3-1 越界错误

(2) {\$R-} 状态，不执行越界检查，这是默认状态。此时不会抛出错误，但是会得到错误的运算结果。请看下面的代码：

```
var
  B: Byte;      {Byte 类型的变量在内存中占据 8 位。使用函数 SizeOf 可以取得一个变量或者一个类型在内存中占据的字节数}
```



```
begin
  B := 255;      {此时 B 在内存中的状态是：11111111 (8 位都是 1)}
  B := B + 1;    {期望获得：100000000 (最高位是 1，其他 8 位是 0)；但是 Byte 最多
                  8 位，所以最高位被抛弃，因而得到低 8 位全 0，也就是最终结果 0}
  ShowMessage(IntToStr(B)); {得到 B=0 而不是 256}
end;
```

Delphi 的 System 单元定义了下列几个全局函数和过程来操作有序变量和有序类型：

Ord 取得有序变量的值在取值范围中所在的顺序（即位置）
Pred 取得有序变量的前序的值（即上一个位置的值）
Succ 取得有序变量的后序的值（即下一个位置的值）
High 取得有序类型/变量的末序的值（即取值范围规定的最大值），或者数组变量的末序
Low 取得有序类型/变量的始序的值（即取值范围规定的最大值），或者数组变量的始序

下面的例子演示了以上几个函数和过程的使用方法：

```
type
  TC = 'A'..'Z';    {定义字符内容的子界类型}
var
  C: TC;
  OI: Integer;
  PI, SI, H, L: Char;
  S: String;
begin
  C := 'B';
  OI := Ord(C);      {OI = 66}
  PI := Pred(C);     {PI = 'A'}
  SI := Succ(C);     {SI = 'C'}
  H := High(TC);     {或者 H := High(C); H = 'Z'}
  L := Low(TC);      {或者 L := Low(C); L = 'A'}
  S :=
    '顺序：' + IntToStr(OI) + #13 +
    '前序：' + PI + #13 +
    '后序：' + SI + #13 +
    '末序：' + H + #13 +
    '始序：' + L;
  ShowMessage(S);
end;
```

因此，对于有序类型的取值规则，可以总结如下：



对于一个初始化后的有序变量 V ，如果它的取值顺序在位置 n ，那么它的前序是 $n-1$ ，后序是 $n+1$ 。如果有序变量 V 已经位于初序或者末序，那么在 $\{R+\}$ 状态，使用 $Pred$ 或者 $Succ$ 将直接返回 $n-1$ 或者 $n+1$ 位置的取值；如果在 $\{R-\}$ 状态（缺省），则 $Pred$ 返回后序取值， $Succ$ 返回前序取值。

好，弄清了简单类型的概念，搞清楚它所属的各种类型就顺利多了，下面来逐一击破之。

Integer 的基本类型是 Integer 和 Cardinal，建议在大多数情况下使用这两种类型，因为它们都是 32 位的，操作系统和 CPU 可以花费最少的时间处理它们。很明显，64 位的 Int64 可以兼容其他所有整数类型，而 8 位的 Byte 则被其他所有整数类型兼容。通常，取值范围大的可以兼容取值范围小的。范围更大的变量也可以赋值给范围小的，但是数据会被“斩断”，这个规则对于实数也是适用的。例如：

```
var
  B: Byte;      {B 是 8 位的}
  W: Word;      {W 是 16 位的}
begin
  W := $1234;   {将十六进制常数$1234 赋值给W}
  B := W;       {B 得到的值是W 的低 8 位，即十六进制的$34，也即十进制的 52}
  ShowMessage(IntToStr(B)); {显示 B 的结果值 52}
end;
```

Character 的基本类型是 Char。其他类型包括 AnsiChar 和 WideChar。AnsiChar 和 Char 是等同的，都是用来处理 8 位字符（即单字节字符），而 WideChar 则是用来处理多字节字符（目前版本的 Delphi 实现为 16 位，即 2 字节字符）的。大家可以参考本小节下面讲解 String 类型的内容，可以对这几个类型有更深入的认识。

Boolean 的基本类型是 Boolean。另外三个是为了兼容操作系统（如一些 API 函数使用）和其他语言（如 C）而定义的。Boolean 和 ByteBool 都是占用 1 个字节，而 WordBool 和 LongBool 分别占据 2 个和 4 个字节。

Enumerated（枚举）类型定义一系列有序值的集合。Enumerated 变量就从这个既定的集合中取某个值。集合中的有序值可以称为元素，元素一般从 0 开始连续索引（也就是元素的顺序号），如：

```
type
  TSize = (Small, Medium, Large);
var
  Size: TSize;
begin
  Size := Large;
  ShowMessage(IntToStr(Ord(Size))); {显示 2}
end;
```

但是也可以显示指定枚举元素的索引（注意这类枚举类型不能生成运行时类型信息），如：

```
type
  TSize = (Small = 5, Medium = 10, BeforLarge, Large = Small + Medium);
var
  Size: TSize;
begin
  Size := BeforLarge;
  ShowMessage(IntToStr(Ord(Size)));      { 显示 11 而不是 2, 因为 Medium 被索引
                                          为 10, 那么其后序相应的为 11 }

  Size := Large;
  ShowMessage(IntToStr(Ord(Size)));      { 显示 15 而不是 3 }
end;
```

Subrange (子界) 类型也是定义一系列有序值的集合, 但是子界的有序值必须是在别的有序类型 (可以将它称为子界的基本类型 (baseType)) 中已经定义过的。子界中有序值的索引被基类确定。如:

```
type
  TColors = (Red, Blue=5, Green, Yellow, Orange, Purple=10, White, Black);
  { 定义一个枚举类型 TColors。TColors 定义了一些有序值供下面的子界类型 TMyColors 使用 }
  TMyColors = Green..White;
var
  MyColors: TMyColors;
begin
  MyColors := Green;      { MyColors 只能在 Green..White 之间取值, 取 Red、Black
                           等是不允许的 }
  ShowMessage(IntToStr(Ord(MyColors))); { 显示 6, 因为元素 Green 在 TColors 中
                                          被索引为 6 }
end;
```

更简单的是我们可以使用 Object Pascal 已经预定义的有序值来定义子界类型, 如:

```
type
  TSomeNumbers = -128..127; { 取值只能在整数 -128 ~ 127 间 }
  TCaps = 'A'..'Z';        { 取值只能在字符 'A' ~ 'Z' 间 }
```

2. Real (实数) 类型

Real 属于 Simple 大类, 但是因为无理数的介入, 使得 Real 不再能够“有序”, 比如 1.0 和 1.2 之间可以取无限个数, 因此, 1.15 的确定顺序就不存在了。

Real 的基本类型是 Real, Real 目前等同于 Double, 但是不排除 Delphi 后续版本修改的可能, 因



此，建议使用 Real 的扩展类型而不是 Real 本身来操作实数。

一般地，声明变量时使用 Double 和 Single，而在过程或者函数中需要定义可以传递任意实数的参数时，常使用 Extended，因为它的取值范围最大，可以兼容其他所有实数类型。

看到这里，不免想起“Float”这个单词。它好像可以用来定义浮点数？不能！在 Pascal 中，Float 只不过是一个概念，代表浮点数的意思，而并不是一个类型标识符，所以不能用它来声明一个变量。如果你在编写开发文档，倒可以考虑让它出场！

3. String (字符串) 类型

看到 String，就好像遇到了老朋友，我们和它打交道实在是太多了。不过如果没有和它促膝长谈过的话，可能就不能深入理解它，因为它的内心还是蛮丰富的。

字符串可以分为三大类：短字符串 (ShortString)、长字符串 (AnsiString / LongString)、宽字符串 (WideString)。

ShortString 在内存中占用 0..255 字节，也就是说，它被固定分配 256 字节。其中第 0 字节存储字符串的实际长度，因此，一个 ShortString 类型的变量实际最多存储 255 个字符；但是，即使实际存储的字符不到 255 个，它仍然在内存中霸道地占据 256 字节。所以为了避免浪费，Object Pascal 又提供了另一种方法来声明更短的 ShortString 变量：String[MaxLength]。一个 String[MaxLength] 类型的 ShortString 占据 0..MaxLength 共 MaxLength+1 个字节。当然 MaxLength 必须小于 256。我们看如下代码：

```
var
  ShortStr: ShortString;
  ShorterStr: String[100];
begin
  ShowMessage(IntToStr(SizeOf(ShortStr)));      { 显示 256 }
  ShowMessage(IntToStr(SizeOf(ShorterStr)));    { 显示 101 }
end;
```

注意：ShortString 标识符是为向后兼容而保留的，新的开发中不要再使用它。在有必要使用短字符串类型时，可以采用“String[MaxLength]”格式来声明。

长字符串和宽字符串的内存是动态分配的，最大可至 2GB，因此，可以近似认为它们是无限长的。

AnsiString 包含的字符是用单字节存储的，而 WideString 包含的字符则是用多个字节存储的。在目前的 Delphi 版本中，WideString 被实现为用 2 个字节存储一个字符。因此，使用 WideString 来处理多字节字符是十分方便的。如：

```
var
  S: AnsiString;
  WS: WideString;
begin
```



```

S := '罗小平';
WS := S;
ShowMessage(S[1]);           { 显示的是乱码, 因为 S[1] 取出的是 '罗' 的一半 }
ShowMessage(WS[1]);          { 显示 '罗' }
end;

```

对于一个长字符串来说, 可以将它看做是一个容纳字符的动态数组, 因此, 下面讲的动态数组的一些特性, 对长字符串来说也是适用的。而一个短字符串相当于一个容纳字符的静态数组, 它占用的内存大小是固定的。

在目前的 Delphi 版本中, String 标识符可能代表 ShortString, 也可能代表 AnsiString。这是由编译指令 \$H 控制的。当在 {\$H+} 状态时, String 等同于 AnsiString, 否则等同于 ShortString。因为 {\$H+} 是默认状态, 所以我们一般都用 String 而不是 AnsiString 来声明长字符串; 如果需要使用短字符串, 就采用 “String[MaxLength]” 格式而不使用过时的标识符 ShortString。

讲字符串类型, 不得不说到 PChar 类型。PChar 并不是 Pascal 的标准类型, 而是为了与操作系统和其他语言兼容而专门设立的一个字符串类型。比如 API 函数 MessageBox 中有两个参数都要求是 PChar 类型。

PChar 声明一个以空字符 (NULL 字符) 结尾的字符串的指针, 更准确地说, 这个字符串的字符是 Char 类型的, 也就是说, PChar 指向的是一个 Char 串。相应地有 PAnsiChar 和 PWideChar, 分别对应于 AnsiChar 串 (近似于 AnsiString 类型。之所以说近似, 是因为 AnsiString 不仅以 NULL 结尾, 而且还在内部存储了串的长度, 但是 AnsiChar 串只是简单地以 NULL 结尾; WideChar 串和 WideString 也是如此。具体可以参考 “3.1.3 数据的内存结构” 小节) 和 WideChar 串 (近似于 WideString 类型)。前面我们说了 Char 和 AnsiChar 在实现上是完全一样的, 所以 PChar 和 PAnsiChar 也是完全等同的。

NULL 是字符串的结束符号, 就像一群人排成几个长蛇阵, 然后每阵的最后那个人打个牌子, 上书 “结束” 二字, 这样可以对不同的阵进行区分。AnsiString 和 WideString 已经以 NULL 结束, 可以实现区分了, 但是它又在内部存储了自己的实际长度, 这样可以在某些操作 (如使用 Length 函数取得字符串长度) 时提高效率, 因为它们存储的字符可能比较多, 如果每次都一个一个字符地数, 会很慢的。

4. Structured (构造) 类型

Structured 可以包含多个元素 (或称字段), 因而这种类型的变量能够容纳多个值。

Set 类型的典型语法如下:

```
set of baseType
```

其中 baseType (基本类型) 可以是 Enumerated 和 Subrange, 因此, Set 变量容纳的值实际上是有序值, 但是在 Set 变量中, 相同的多个有序值是没有意义的, 被当作一个处理。比如:

```

type
  TOneSet = set of (A, B, C);
var

```



```

OneSet: TOneSet;
begin
  OneSet := [A];
  OneSet := OneSet + [A];
  { 或者 Include(OneSet, A); 不要以为此时 OneSet 中有两个 A 元素, 就好像我已经吃
    饱了, 再给我一碗饭但是我吃不下去 }
  OneSet := OneSet - [A];
  { 或者 Exclude(OneSet, A); 此时 OneSet=[] 了, 不包含任何元素 }
  if OneSet = [] then
    ShowMessage('OK');
end;

```

Array (数组) 是我们再熟悉不过的数据类型了, 数组有两种形式: 静态数组和动态数组。动态数组要首先使用 SetLength 分配用来存放元素的空间后才能使用。值得一提的是, 如果使用 SetLength 给动态数组重新分配空间, 那么已有元素可以得到保留。如:

```

type
  TOneArray = Array of Integer;
procedure AddArrayItem(A: TOneArray; I: Integer); { 给动态数组增加一个元素 }
begin
  SetLength(A, Length(A)+1); { 可以使已有元素得到保留 }
  A[High(A)] := I;
end;

```

Record (记录, 有的语言中称为结构 (Structure)) 类型, 是多个任意元素的集合, 其中的元素可以称为记录的字段。因为本书在很多地方使用了记录类型, 所以在这里就不浪费纸张作展开说明了。在记录类型中, 难一点的是变体记录 (主要目的是用来实现数据共享), 将放在 3.1.4 节中讨论。

File (文件) 类型用来读写文件。不过在面向对象编程中, 我们更倾向于使用 TFileStream 类和一些类 (如 TStrings、TPicture、TBlobField、TTreeView 等) 的 LoadFromFile、SaveToFile 方法来操作文件。

好了, 构造类型还剩下 Class (类)、Interface (接口) 和 Class reference (类引用) 三种子类型。Class 和 Interface 牵涉的方方面面很多, 不是三言两语可以说得清的, 而 Class reference 则是非常抽象的概念, 所以我们必须在后面开辟专门的章节来讲解它们。本书的具体安排是: Class 和 Interface 放在 3.3 节; Class reference 放在第 9 章, 以 9.4 节结合具体例子来阐述, 不过在该节之前, 本书会无法避免地多次提到类引用, 因此, 朋友们读到这些地方, 如果对类引用有不明白的地方可以跳读第 9 章的内容。

5. Pointer (指针) 类型

一个指针占用 4 字节空间, 或者说, 一个指针就是一个 4 字节大小的内存块。该内存块的 4 字节

空间用来存储另一块内存区所在的地址，这另一块内存区才是存储实际数据的地方。打个比方，如果你有女朋友的话，你们上街时，你们之间就有个指针了——那就是女朋友挽着你胳膊的纤纤玉手；你女朋友看到漂亮的衣服时，就会通过这个指针找到你，并让你掏钱买单。

那么程序呢，也是通过指针找到它所指的地址，来存取具体内容；如果实际内容是划分为多个域的（比如一个对象包含多个字段和方法，分别存放在不同的地址域里），那么这个指针是指向该域群的首地址（就像女朋友挽着你的胳膊或者揪住你的耳朵），程序根据每个域对于首地址的偏移量来寻址，从而找到特定域的地址，再存取具体内容。

大多数数据类型（包括指针自己）都可以有对应的指针类型。如 PInteger、PBoolean、PString、PVariant、PPointer 等都是 Object Pascal 预定义的指针类型。在 Delphi 中定义一个指针类型的典型语法是下面这个样子的：

```
type PointerTypeName = ^type
```

如 PInteger 和 PPointer 是这样定义的：

```
PInteger      = ^Integer;      { 定义指向 Integer 类型的指针 }
PPointer      = ^Pointer;      { 定义指向指针的指针 }
```

指针可以分为两大类：无类型指针（Untyped Pointer）和有类型指针（Typed Pointer）。

直接用 Pointer 声明的变量即是无类型指针，可以在使用时指向任何数据类型；它好像是还没有男朋友的姑娘的手，以后会揪住谁的耳朵，目前还是未知的；也类似于 TObject，可以代表任何类型的对象。有类型指针所能指向的数据是固定类型的，至少必须是兼容的，比如 PInteger 不能指向一个字符串，但可以指向一个 Byte 或者 Word 变量。

指针的常用操作符是@（或者函数 Addr，取得地址，通常用于给指针变量赋值）和^（从指针变量取得实际的数据）。如：

```
type
  TOneRecord = record
    Name: String;
    Age: Word;
  end;
  POneRecord = ^TOneRecord;
var
  OneRecord: TOneRecord;
  PRecord: POneRecord;
begin
  OneRecord.Name := '罗小平';
  OneRecord.Age := 25;
```




```

PRecord := @OneRecord;    { 让 PRecord 指向 OneRecord }
PRecord.Name := '罗小平';
{ 你会看到以下两句代码显示的结果是相同的, 也就是可以通过指针直接存取实际数据。从理
论上看"PRecord.Name" 这种形式的引用是不合法的, 实际上是编译器帮助你将"PRecord.Name"
处理成了"PRecord^.Name"。所以表面上的不合法最终被改造成了合法 }
ShowMessage(PRecord^.Name);
ShowMessage(PRecord.Name);
end;
```

Delphi 中的很多数据类型在内部都是实现为一个指针的, 尽管它们在表面上看起来不是这样。比如类和接口的实例本身就是指针, 还有如长字符串、动态数组、类引用等实际上也是指针, 它们内部都是通过指针来实际实现数据存取的, 具体可参看本节后面的内容。如果你觉得这种说法有点不可思议, 那么到了冬天, 你被女朋友戴着手套的手拽到商店时, 你可能就感觉豁然开朗了——原来戴不戴手套结果都是一样的。一般情况下是不需要声明类实例等的指针的, 比如下面的代码:

```

procedure TForm1.Button1Click(Sender: TObject);
type
  PObject = ^TObject;
var
  PObj: PObject;
begin
  PObj := @Self;
  ShowMessage(PObj^.ClassName); { 的确没有错, 显示了"TForm1", 但是没有必
                                要定义PObject 这个类型。可行的并不总是必要的 }
end;
```

我们可以用一个简单的方法来判断某个类型或者变量实际上是否是一个指针: 如果 SizeOf(一个类型或者变量) 返回 (返回值是该数据类型要占据的内存大小, 以字节为单位) 4, 而这个类型或者变量的实际数据又并不是 4 个字节空间可以完全存储的, 那么此时它很可能是一个指针。比如:

```

var
  { 以下四个变量的数据显然都不是 4 个字节可以完全存储的 }
  A: Array[0..1] of Integer;
  DA: Array of Integer;
  SS: String[10];
  S: String;
{ 我们这里将 ShowInfo 定义为一个子过程。一个子函数或者子过程的作用范围在母函数或者
母过程内部、其他地方不能使用。这种定义局部函数和过程的方法在 Delphi 开发中经常
使用 }
```



```

procedure ShowInfo(Obj: String);
begin
    ShowMessage(Obj + '实际是一个指针。');
end;
begin
    SetLength(DA, 2);
    if SizeOf(A) = 4 then
        ShowInfo('A');
    if SizeOf(DA) = 4 then      { 结果为 True , 表明动态数组实际是一个指针 }
        ShowInfo('DA');
    if SizeOf(SS) = 4 then
        ShowInfo('SS');
    if SizeOf(S) = 4 then      { 结果为 True , 表明长字符串实际是一个指针 }
        ShowInfo('S');
end;

```

费了这么多篇幅来说指针,可以套用阿基米德的一句话:“给我四个字节,我就可以操纵整个地球”。意思是说指针的能耐很大、地位很重要,所以我们不得不花大力气搞明白它。

6. Procedural (过程) 类型

过程类型算是一种比较特殊的类型。这种类型的变量可以存取一个过程或者函数!是不是有点难以想象?“过程类型”类型的“过程”一词是广义的,包括过程和函数。

其实我们和过程类型打交道的时候很多,是面熟而不知其名罢了。我们在 Object Inspector 中随便看一个按钮的 OnClick。OnClick 是什么,是一个单击事件,后面的章节我们会讲,事件其实是一种特殊的属性。那么 OnClick 既然是属性,它肯定是属于某种数据类型的,这个数据类型是什么呢?在 OnClick 上按 F1 键,我们在帮助里看到:

```
property OnClick: TNotifyEvent;
```

也就是说,OnClick 是 TNotifyEvent 类型的数据。再看 TNotifyEvent,发现是这样一种类型:

```
type TNotifyEvent = procedure (Sender: TObject) of object;
```

看起来像个怪胎。怪胎的真实名字叫做过程类型。因为 TNotifyEvent 后面加了“of object”关键字,所以更准确地讲,TNotifyEvent 是一个方法类型。

使用下面的语法可以定义一个过程类型:

```

type
    TMyFunction = function (参数列表): 返回类型;
    TMyProcedure = procedure (参数列表);

```

如果在后面加上了“of object”关键字,那么它就成了一个特殊的过程类型:方法类型。一个方法



类型是隶属于对象的过程类型。

过程类型的变量可以当作普通变量一样使用，但更重要的用途是实现一个过程调用，或者作为参数传递，从而实现回调函数功能。看下面一段代码：

```

type
    TOneFun = function(X: Integer): Integer;      { 声明一个过程类型 }

function SomeFunction(X: Integer): Integer;
{ 实现一个和 TOneFun 兼容的过程 }
begin
    Result := X*2;
end;

function SomeCallBack(X: Integer; OneFun: TOneFun): Integer;
{ SomeCallBack 被调用时，回调函数 OneFun，并返回 OneFun 的执行结果 }
begin
    Result := OneFun(X);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    F: TOneFun;
{ 声明过程类型变量，也可以直接声明：F: function(X: Integer): Integer; }
    I, J: Integer;
begin
    F := SomeFunction;      { 用过程类型变量 F 引用一个实际过程 }
    I := F(4);              { 通过过程类型变量 F 直接调用函数 SomeFunction }
    J := SomeCallBack(4, F); { 通过过程类型变量 F 回调函数 SomeFunction }
    if I = J then
        ShowMessage('F(4) 和 SomeCallBack 功能相同');
end;

```

运行上面的代码，我们发现 I 和 J 最终是相等的。

过程类型的变量实际上是指针，可以称为过程指针，这个指针指向某个过程或者函数所在地址；相应地，方法类型的变量可以称为方法指针，它当然是指向某个方法所在地址。方法类型在 VCL 中大量使用，一个事件实际上就是一个方法类型的属性，也就是说是一个方法指针。

从表面上看，过程类型和方法类型的区别在于“of object”关键字，而造成方法类型的变量只能通过对象来引用。实际上，方法类型除了指向方法地址的指针外，还有一个附带的指针——指向所属的

对象。因此，普通过程类型和方法类型是不兼容的，不能直接相互赋值。

方法指针可以用定义在 System 单元的一个记录描述：

```
type
  TMethod = record
    Code, Data: Pointer;
  end;
```

我们可以看到，它包含两个指针 Code 和 Data。Code 可以看做是方法地址指针，Data 可以看做是方法所属对象的指针。我们看下面一个例子：

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  published
    procedure ShowInfo;
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.ShowInfo;
begin
  ShowMessage(Self.Name);
end;

procedure TForm1.Button1Click(Sender: TObject);
type
  TMyProcedure = procedure of object;
var
  OneProcedure: TMyProcedure;
begin
  OneProcedure := Form1.ShowInfo; { 给方法指针赋值 }
  ShowMessage(TObject(TMethod(OneProcedure).Data).ClassName);
```

```

{ 显示'TForm1' }
  OneProcedure; { 显示'Form1' }
end;

```

由于我们将 ShowInfo 声明在 published 域，所以给方法指针赋值（即“OneProcedure := Form1.ShowInfo;”），也可以用下面两行代码代替：

```

TMethod(OneProcedure).Code := Form1.MethodAddress('ShowInfo');
TMethod(OneProcedure).Data := Form1;

```

必须强调的是，方法指针实际上包含两个指针，所以如果你去掉上面代码的第二行（即给 Data 赋值的那行），那么最后的 OneProcedure 调用将发生异常。这是因为“ShowMessage(Self.Name);”希望引用方法所属的对象，而该对象实际上就是 Data 指针，但因为没有给 Data 赋值，因此它为 nil。

注意：判断一个过程类型变量是否存在实际值，不能用“<> nil”这样的格式直接比较，而应该使用函数：

```
function Assigned(const P): Boolean;
```

函数 Assigned 测试一个指针或者过程类型变量是否为 nil。

7. Variant（可变）类型

关于 Variant 我们需要理解以下几点：

（1）Variant 可以存储绝大部分不同类型的数据（但是指针类型数据只能用 PVariant 来存储）。

（2）Variant 变量在某时刻有三个可能的状态：Unassigned（表示没有值，可以用函数 VarIsEmpty 来测试）、Null（值为 NULL，可用函数 VarIsNull 测试）和非 NULL 值。声明一个 Variant 变量后，它被置为 Unassigned 状态。

这三种状态可以说是 Variant 的三张脸，看它的脸色行事是十分重要的；否则它不会给你好脸看。最重要的是记住不要对 Null 状态的 Variant 变量进行操作，否则会抛出 EVariantInvalidOpError 类型的错误。比如：

```

var
  V: Variant;
begin
  V := NULL;
  V := V + 1;
  ShowMessage(V);
end;

```

上面的代码必然抛出异常，没有商量的余地。

（3）我们可用函数 VarType 来判断 Variant 变量所存储数据的实际类型。VarType 返回一个 Word 类型数据，System 单元定义了一些常数用来代表这个返回值，如 varEmpty（在 Unassigned 状态）、varNull

(在 Null 状态) varInteger (数据为 Integer 类型) 等。

大多数时候, VarType 都能返回一个简单的类型常数。但是对于一些特殊的变量,如数组、引用,那么 VarType 会返回一个复合的类型常数:前 12 位(0~11)表示简单数据类型,第 14 位、15 位分别表示是否是数组(varArray(\$2000)) 引用(varByRef(\$4000))类型。我们用常数 varTypeMask(\$0FFF)可以分离这两个部分。例如:

```
var
  V: Variant;
begin
  V := VarArrayCreate([1,2], varInteger);
  ShowMessage(IntToHex(VarType(V),2)); {显示 2003,即 varInteger+varArray}
  ShowMessage(IntToHex(varTypeMask and VarType(V),2));
  {显示 03,即 varInteger}
end;
```

编译器在进行 Variant 类型转化时,首先使用 VarType 判断其数据类型,然后根据内定的规则转化。

(4) Variant 的扩展类型 OleVariant。Variant 类型的数据只能在同一个应用程序中传递,当需要在不同应用程序、不同计算机间传送 Variant 数据时,需要使用 OleVariant。

在本小节的最后,我们不得不讲讲“类型别名”这个话题。

下面是从 Windows 和 Types 单元抽出的两行代码:

```
(1)
type
  DWORD = LongWord;

(2)
type
  HWND = type LongWord;
```

以上就是定义类型别名的两种方法。

第一种方法定义一个 LongWord 类型的别名 DWORD,在任何时候 DWORD 和 LongWord 都是兼容的,因为它们实际上是同样的类型,只是名字不同而已。

第二种方法定义一个 LongWord 类型的新别名 HWND。如果你读到这里想打瞌睡的话,那么希望你赶紧将大腿拧一把醒过来。要注意:此时的 HWND 和 LongWord 是完全不同的两种类型,尽管它们在内存中的存储方式没有任何差别!在简单赋值时,编译器认为 HWND 和 LongWord 是兼容的,但是用于 var 和 out 参数等要求类型严格匹配的地方时,则被认为不兼容。

Delphi 提供新别名的主要用途在于实现一些特定用途的属性编辑器。在本书后面讲属性编译器时,我们就定义了 String 的新别名,以便给 TFilePathName 实现一个属性编辑器:



```
type
  TFilePathName = type String;
```

在程序编写中，要尽可能选择简单、轻量级、易懂的数据类型，除非实际需要，不要使用复杂的数据类型或者自定义类型，不要故作高深、故弄玄虚，须知最简单的才是最美丽的。比如要定义整数类型时，要尽可能使用 Integer 和 Cardinal 这两个基本类型。

湖北有道美食：“三耳猴头”，是用神农架出产的猴头菌、兴山的血耳、恩施的石耳和房县的桂花耳加冰糖炖。你看，光是找材料就够烦人的了，更不用说它的做法。日常生活中，我们实在没必要吃这个菜。

小结

本小节全面讲述了 Object Pascal 中的数据类型，并提供一张数据类型全家福来归附它们的相互关系，为程序开发时数据类型的选择提供了重要参考。

3.1.2 变量的内存分配和释放

从作用范围的角度，变量可以分为两大类：全局（Global）变量和局部（Local）变量。

函数或者过程内部定义的变量为局部变量；其他的变量被声明在 interface 和 implementation 部分，称作全局变量，可以在整个单元中引用。对于在类中声明的变量，如果我将类比作单元，那么类中的变量可以比作单元中的全局变量；类的方法中声明的变量可以比作函数和过程中的局部变量。以下所讲的内存分配形式对于类中的变量也是适用的。

变量的内存分配形式有两种：自动和人工。所谓自动分配，是一个变量被声明后即被分配内存；而人工分配是指变量被声明后必须用代码显式地分配内存。

一般地，无论是全局变量还是局部变量，如果它是非指针类型的，则声明后被自动分配内存。如果是全局变量，还会被初始化为 0：数值类型的为 0，布尔类型的为 False，字符的为 ”，等等。如果是局部变量，则不会被初始化，因此，它的值是不确定的（取决于别的程序对这块内存作过的操作）。对于非 Variant 和 File 类型的全局变量，还可以在声明时指定初始值（如：var I: Integer = 7;），但是对任何的局部变量都不可以这么做。

如果变量是指针类型的，则不会被自动分配内存。如果它是全局的，则其初始值是 nil，表示还没有指向；如果是局部的，尽管没有被分配内存，但是会随机地指向一个地址，因此值不是 nil。

为了验证上述内容，我需要举几个例子。

例 1 验证全局变量的内存分配形式：

```
var
  Global_Int: Integer; {声明非指针类型的全局变量 Global_Int}
  Global_P: PChar;    {声明指针类型的全局变量 Global_P}
```

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    if @Global_Int <> nil then
    {用@Global_Int 取得Global_Int 的地址指针, 然后和nil 比较。此条件为True}
    begin
        ShowMessage('Global_Int 已被分配内存');
        ShowMessage(IntToStr(Global_Int));           {显示初始值0}
    end;
    if Global_P = nil then                           {此条件也为True}
    begin
        ShowMessage('Global_P 还没被分配内存');
        ShowMessage(Global_P);
    {期望显示Global_P 指向地址处保存的字符串, 但因为指向不存在, 所以返回空字符串''}
    end;
end;

```

例 2 验证局部变量的内存分配形式：

```

procedure TForm1.Button1Click(Sender: TObject);
var
    Local_Int, Local_Int2: Integer;
    OldAddr, NewAddr: Integer;
    Local_P: PChar;
begin
    OldAddr := Integer(@Local_Int); {取得Local_Int 声明后的地址}
    Local_Int := 7;                 {给Local_Int 赋值}
    NewAddr := Integer(@Local_Int); {取得Local_Int 被赋值后的地址}
    if OldAddr = NewAddr then      {这个条件为True}
        ShowMessage ('地址值没有变化, 所以声明 Local_Int 时就分配内存');
    ShowMessage(IntToStr(Local_Int2)); {非指针局部变量的初始值不是0}
    if Local_P <> nil then         {这个条件也为True}
        ShowMessage(Local_P);      {显示的不是空字符串而很可能是乱码}
end;

```

综上所述，无论是全局变量还是局部变量，非指针类型的变量是被自动分配内存的，这个工作由编译器编译时完成，所以这种分配方式也被称作静态分配。静态分配时，全局变量的内存分配在全局变量区，局部变量分配在应用程序栈（Stack）。它们的内存释放工作也被自动管理，不需要程序员干预。

注意：应用程序可用的内存区分为三类：全局变量区（专门用来存放全局变量）、栈（Stack）和堆（Heap）。应用程序开始运行时，所有全局变量的内存被分配到全局变量区，应用程序结束时被释放；被分配在栈上的变量内存可被栈管理器自动释放；堆上的变量内存必须人工释放。

一般而言,对于指针类变量,则需要程序员使用一些代码来完成内存分配,通常,这样的分配方式也被称作动态分配。但是也有一些指针类型的变量是被动态分配内存的,它们是:

长字符串(AnsiString/String)、宽字符串(WideString)、动态数组(dynamic arrays)和接口(interface),这些类型的变量也是被自动释放内存的。动态数组和接口也可以人工释放内存,方法是赋值 nil。

完成具体分配的方法主要有下列一些:

(1) 赋值。其原理是将变量指向一块已经存在的内存。这种方法适用于所有的指针类型。比如:

```
var
  P1, P2: PChar;
begin
  P1 := 'lxpbuaa'; {P1 已经拥有一块内存}
  P2 := P1;        {将P2 指向P1 的内存,这样就间接完成了P2 的内存分配}
end;
```

这种方法的本质是多个指针共享一块已有的内存,因此,通过操作任何一个指针都可以达到内存释放的目的。如果该块内存是被自动管理的,那么就不需要人工释放。

(2) 对于类,则调用构造函数。比如:

```
var
  Obj: TObject;
begin
  Obj := TObject.Create; {调用构造函数创建对象,变量Obj 指向该对象}
  Obj.Free;              {释放内存,Free 内部调用析构函数 Destroy; 也可以使用
                          FreeAndNil(Obj);}
end;
```

对象变量所指向的内存是必须人工释放的,因为该块内存被分配在堆(Heap)而不是栈(Stack)上。释放对象内存时,应该调用析构函数(通常调用析构函数的包装方法 Free 或者全局过程 FreeAndNil 即可)。

(3) 分配指定大小的内存块。主要用于创建缓冲区,一些函数和过程通过缓冲区返回一些执行结果。比如文件读写、流读写以及大量的 API 函数。我们看一个 API 函数使用缓冲区的例子,该函数可以取得计算机名字:

```
var
  P: PChar;
  Size: Cardinal;
begin
  Size := MAX_COMPUTERNAME_LENGTH + 1;
```

```

GetMem(P, Size);           { 分配 Size 个字节的内存块 (即缓冲区), 并让 P 指向它 }
GetComputerName(P, Size); { API 函数 GetComputerName 将取得的计算机名放在 P
                           中 }

ShowMessage(P);

FreeMem(P);                { 释放缓冲区占用内存 }

end;

```

在上例中, 我使用了 GetMem 过程来创建内存块, 并用 FreeMem 来释放它。总结一下, 动态分配内存的函数和过程有以下一些, 它们都是在堆中分配内存, 所以必须释放:

(1) GetMem:

```
procedure GetMem(var P: Pointer; Size: Integer);
```

分配大小为 Size 字节的内存块, 并让 P 指向它。

(2) AllocMem:

```
function AllocMem(Size: Cardinal): Pointer;
```

分配大小为 Size 字节的内存块并初始化为零, 并返回地址指针。

如果希望在中途改变先前用 GetMem 或者 AllocMem 分配的内存大小, 可以使用 ReallocMem:

```
procedure ReallocMem(var P: Pointer; Size: Integer);
```

使用 GetMem 和 AllocMem 分配的内存都应该用 FreeMem 释放:

```
procedure FreeMem(var P: Pointer);
```

(3) New:

```
procedure New(var P: Pointer);
```

用 New 分配的内存块大小由参数 P 的类型确定, 因此, 不要使用它给无类型指针 (即 Pointer 类型) 变量分配内存。释放该内存块时使用 Dispose:

```
procedure Dispose(var P: Pointer);
```

小结

本小节详细讨论了变量内存分配的两种形式。重点是:

- (1) 全局变量和局部变量的内存分配异同。
- (2) 变量内存分配和释放什么时候是自动/人工的。
- (3) 如何人工分配和释放变量内存。

3.1.3 数据的内存结构

前面的一些地方提到：一些数据类型在内存中的实际存储方式，并不是表面上看起来那样子的。比如一个动态数组，并不是简单地在内存开辟一个区域，然后按地址顺序逐个放置数组的元素，长字符串也不是字符们脑门贴脑勺挤成一个长蛇阵那么简单。

在这个小节里，我们将分析一些数据类型变量在内存中的真实存放格式，从而了解这些类型的真实运作方式。不同平台对于数据的内存管理方式是有所不同的，那么本书呢，当然都是指 Windows 平台。

1. Boolean 类型

基本类型 Boolean 变量被存储为一个 Byte，数值为 0 代表 False，为 1 代表 True。

其他扩展类型是为了和操作系统与其他语言兼容而设立。ByteBool 存储为一个 Byte，WordBool 存储为一个 Word，LongBool 存储为一个 LongInt。数值 0 代表 False，非 0 则表示 True。

2. Enumerated 类型

它的存储格式和编译指令 \$Z 相关。\$Z1(默认)、\$Z2、\$Z4 分别表示被存储为 Byte、Word、DWORD (即 LongWord)。也就是说，枚举变量其实是一个整数，根据这个整数值可以取得不同的枚举值。比如整数为 5 时，那么就表示枚举值应该是原始集合的第 5 个有序值。

所以从上面也可以看到，集合中有序值的最大个数是随 \$Z 变化的。在默认情况 (\$Z1) 下，最多只能枚举 256 个有序值。

3. AnsiString/String 类型

它内部包含四个域：

偏移/Byte 内容

-8 存储引用计数；

-4 存储字符长度；

0..Length-1 存储实际字符；

Length 零字符 (NULL 或者 #0)。

可见，一个 String 变量实际上是一个指针，指向第一个字符所在位置 (即偏移 0 处)。

引用计数域帮助管理内存的自动释放工作。对于一个字符串常数，引用计数总是 -1。

零字符域便于和 PChar 类型的转化。

WideString 变量是类似的，但是少了引用计数域。

在这里我们也发现，AnsiString 和 WideString 都用 4 个 Byte (即 32Bits) 来存储字符长度，而 32Bits 所能表示的最大整数是 2147483647 (即 2G)，所以它们的最大字符长度为 2GB。当然，以后的 Delphi 版本是可以扩展这个长度的。

知道了 String 的实际存储结构，我们就可用下面的代码取得字符串的长度 (注意这样做只是为了说明原理，实际编程则没必要搞得这么复杂，直接使用函数 Length 即可)：



```
var
  S: String;
  L: Integer;
begin
  S := 'lxpbuaa';
  L := PInteger(Integer(S) - 4)^;
  { 或者 L := PInteger(PInteger(@S)^ - 4)^; }
  { 或者 L := PInteger(PInteger(Addr(S))^ - 4)^; }
  { 最终得到 L = Length(S) = 7; }
end;
```

4. Set 类型

集合实际是一个 Bit 数组, 每个 Bit 分别用 0/1 表示是否包含对应元素。因为 Delphi 规定了集合类型的元素不能超过 256 个, 所以一个集合变量最多占用 256 个 Bit (即 8 个字节)。

集合总被分配最小字节个数的内存, 比如有 6 个元素的被分配 1 个字节, 有 9 个元素的被分配 2 个字节。

5. Dynamic array 类型

动态数组内存也被划分为几个域:

偏移/Byte	内容
-8	存储引用计数
-4	存储元素数目
0..Length * (元素的大小) - 1	存储元素值

它和长字符串的存储是十分类似的, 所以大家一看就明白, 我没必要再多说了。

6. Variant 类型

Variant 内部存储为 TVarData 类型的记录。TVarData 被定义在 System 单元。

该记录主要包含两个字段:

- (1) VType (TvarType 类型), 它用来存储数据的类型。在上一小节我们已经讲过这个类型。
- (2) 另一个字段为 8 字节大小, 用来存储实际数据或者指向该实际数据的指针。

所以, 根据这两个字段, 一个 Variant 可以和其他数据类型相互转化。

小结

本小节揭示了数据类型在底层的管理方法, 剖析了一些常见类型的数据在内存中的存储格式和信息控制方法, 有利于我们深入认识数据类型的设计和运作机理。

3.1.4 强数据类型与类型转化

Object Pascal 是一种“强数据类型”语言。



所谓“强数据类型”，就是严格区分不同数据类型，并不总是允许不同类型数据直接赋值。例如：

```
var
  B: Boolean;
begin
  B := 1;
end;
```

上面的代码期望将整数直接赋值给一个布尔变量，这在 Object Pascal 中是行不通的。编译上面的程序时，编译器会生成如下错误信息：

[Error] Unit1.pas(29): Incompatible types: 'Boolean' and 'Integer'

上述信息表示 Boolean 和 Integer 类型不兼容。

但是大家知道在 C 等语言中，是可以将一个整型数赋值给一个布尔变量的。

Pascal 以语法严谨而著称，强类型是其严谨的一个重要体现。但是强类型特性也不是为了体现严谨而刻意加上去的，如果那么搞的话，恐怕今天也不会有这么多人使用 Delphi 了。强类型是有好处的，那就是：可以避免运行时不同类型数据转化出现错误，因为在程序被编译时，潜在的错误就已经被消灭了。

如果说世界上有绝对完美的东西，大家都是不相信的，就像无法找到满足聪明、漂亮、脾气好等 100 个条件的女朋友那样，那只能是做梦。强数据类型尽管有它的好处，但是在不同数据类型相互转化时则带来了很大的障碍，显得非常死板。而数据类型转化对于一个程序员来说，就像肉食对我那样重要，每天都不可缺少。为此，Object Pascal 又特意提供了一些数据类型转化方法。归纳起来，有以下一些：

typecasting, pointers, variants, variant parts in records, absolute addressing

1. typecasting (类型强制转化)

顾名思义，typecasting 是将一个类型的数据强制转化为另一个类型的数据。其典型语法如下：

typeIdentifier(expression)

其中 typeIdentifier 是结果类型标识符，expression 可以是原始数据表达式或者变量。如：

```
var
  B: Boolean;
begin
  B := Boolean (1); { 将整数 1 强制转化为布尔类型，结果 B=True }
end;
```

对于对象和接口，除了使用 typeIdentifier 外，Object Pascal 还专门提供了 as 操作符进行转化。在使用 as 前应该首先判断源对象和结果类型是否兼容、源对象/源接口是否支持结果接口，可分别使用 is 操作符和 Supports 函数来作这种判断。

2. pointers (指针)

首先定义一个结果类型的指针，然后将该指针指向原始表达式（使用操作符@或者函数 Addr），然后从指针所指地址中取出结果类型的数据（使用操作符^）。如：

```
var
  I: Integer;
  P: PBoolean;
  B: Boolean;
begin
  I := 1;
  P := @I;
  B := P^;      { 此时 B=True }
end;
```

3. Variants (可变类型)

Variant 可以存储任何类型的表达式，因此，通过 Variant 这样一个中转站，可以实现数据类型转化。如：

```
var
  V: Variant;
  B: Boolean;
begin
  V := 1;
  B := V;      { 通过 Variant 实现类型转化，此时 B=True }
end;
```

4. variant parts in records (变体记录)

一个变体记录典型的定义如下：

```
type recordTypeName = record
  fieldList1: type1;
  .....
  fieldListn: typen;
case tag: ordinalType of
  constantList1: (variant1);
  .....
  constantListn: (variantn);
end;
```



其中 case 到结尾部分定义了多个变体字段。所有变体字段共享一段内存，换句话说，如果你给 constantList1 赋值，那么 constantList2-constantListn 也就被赋了值。至于这些变体字段返回什么值，则是由它们的类型决定。程序根据 tag 的值决定应该使用 constantList1-constantListn 中的哪个字段。例如：

```
type
  TDataConv = record
    case T: Boolean of
      True: (I: Byte);
      False: (B: Boolean);
    end;

var
  D: TDataConv;
begin
  D.I := 1; {此时 D.B=True, 因为 I 和 B 这两个变体字段共享一段内存}
end;
```

使用变体记录时要注意：

(1) 变体字段不能是 long strings、dynamic arrays、variants、interfaces 等由编译器自动管理内存的类型，也不能是含有上述类型的构造类型，但可以是这些类型和构造类型的指针。

(2) 所有变体字段共享一段内存。而共享内存的大小则由最大变体字段决定。

(3) 当 tag 存在时，它也是记录的一个字段。也可以没有 tag。

再看 Messages 单元定义的消息类型 TMessage：

```
TMessage = packed record
  Msg: Cardinal;
  case Integer of
    0: (
      WParam: Longint;
      LParam: Longint;
      Result: Longint);
    1: (
      WParamLo: Word;
      WParamHi: Word;
      LParamLo: Word;
      LParamHi: Word;
      ResultLo: Word;
      ResultHi: Word);
  end;
```



case 部分并没有 tag 字段,接下来的 0 和 1 只是为了给变体字段分组,0 部分的三个字段和 1 部分的六个字段共享一段内存。这段内存大小是 4 (Longint 即 Integer, 占用 4 个字节) × 3 = 12 个字节。一个 Word 占用 2 个字节。我们知道一个 32 位整数在内存中是高字节在后,低字节在前,因此, WParamLo 被对应到 WParam 的低 16 位, WParamHi 被对应到 WParam 的高 16 位, 依此类推。换句话说,通过 WParamLo 可以取得 WParam 的低 16 位,而通过 WParamHi 可以取得 WParam 的高 16 位。

5. absolute addressing (绝对地址)

声明共享一段内存的多个变量,从而实现类型转化。和变体字段的机理相似。

一般都很少使用绝对地址来实现数据共享和类型转化,因为它在声明时就固定了变量的地址,导致灵活性缺乏。

下面一段代码演示了如何使用绝对地址来共享数据:

```
var
  I: Integer;
  B: Boolean absolute I; {其中 absolute 是共享关键字,它让 B 和 I 共享一段内存}
begin
  I := 1; {此时 B=True}
end;
```

上面所讲的各种类型转化方法,本质上都是通过数据的内存地址操作来实现的,因此,理解这些转化方法的关键还是在于理解地址、指针这些概念。

除了上面列出的几种转化方法以外, Delphi 也提供了大量的函数和方法来直接实现特定的数据类型转化。关于这些函数和过程的详细列表和功能介绍,请参看“8.1 数据类型转化类”。

在本小节最后,我想举一个较全面的例子,来加深我们对类型转化的认识。这个例子是说非整数类型数据如何在消息中传递。

对于以下两个常用的发送消息 API 函数:

```
function PostMessage(hWnd: HWND; Msg: UINT;
  wParam: WPARAM; lParam: LPARAM): BOOL; stdcall;
function SendMessage(hWnd: HWND; Msg: UINT;
  wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;
```

我们知道其中各个参数含义如下:

hWnd: 接受该消息的窗口的句柄;

Msg: 消息代号;

wParam、lParam: 第一个、第二个消息数据。其类型 WPARAM = LongInt = Integer。

如果我们需要通过 wParam、lParam 来传递非整数类型的数据,如一个记录、字符串甚至对象时,



应该怎么办呢？这两个参数只能是 Integer 类型啊！

很显然，这时候我们要将非整数类型的数据转化为一个整数，然后才能作为参数传递。消息接收方收到消息，再将整数类型的消息参数还原为原始类型。

怎么实现这种数据类型转化呢？那就是传地址。即使用消息参数传递数据变量的地址，而地址是用整数表示的。

比如 Windows 标准消息 WM_GETTEXT 用于取得一个窗口的文本，下面我们就利用它来取得一个 Edit 的 Text 属性。

WM_GETTEXT 需要参数 lParam 作为存储返回结果的缓冲区，wParam 指定要求取得数据的最大长度。于是我们可以写出如下代码：

```
var
  PText: PChar;
begin
  GetMem(PText, MAXBYTE);
  { 注意下面一行的 Integer(PText), 它取得变量 PText 的地址作为参数传送 }
  SendMessage(Edit1.Handle, WM_GETTEXT, MAXBYTE, Integer(PText));
  ShowMessage(PText);
  FreeMem(PText);
end;
```

接收消息时需要通过地址取得原始数据。下面我们利用消息 WM_COPYDATA 举个完整例子：
发送消息：

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Data: CopyDataStruct;
  MsgText: PChar;
begin
  MsgText := 'lxpbuaa';
  with Data do
  begin
    dwData := 100;
    lpData := MsgText;
    cbData := Length(MsgText);
  end;
  SendMessage(Handle, WM_COPYDATA, Handle, Integer(@Data));
end;
```



接收消息：

```
{ 首先声明一个消息方法映射消息：}
procedure WMCopydata(var Msg: TMessage); message WM_COPYDATA;

{ 实现该消息方法：}
procedure TForm1.WMCopydata(var Msg: TMessage);
var
    Data: CopyDataStruct;
    S: String;
begin
    { 因为传过来是数据的地址，所以我们首先用 CopyDataStruct 的指针类型 PCopyDataStruct
      将地址转化为一个指针，再用^操作符取得指针所指实际内容}
    Data := PCopyDataStruct(Msg.LParam)^;
    with Data do
    begin
        S := '数据类别代码：' + IntToStr(dwData) + #13 +
            '消息实际数据：' + PChar(lpData);

        end;
        ShowMessage(S);
    end;
```

小结

本小节描述了 Pascal 强类型特性的优点和缺点，并全面介绍了克服该缺点的几种方法。熟练掌握这些方法，可以大大提升在编写程序时调度各种数据类型的能力。

3.2 过程和函数

过程和函数就是用来完成特定功能的一些代码组成的程序块。它就好像是公共汽车，我们乘坐公共汽车，可以到达目的地。如果坐一路公共汽车还不能到达终点，我们还可以在中途换乘，就相当于调用多个过程和函数来完成一项功能。

过程和函数的惟一区别就是：函数有返回值而过程没有。

在 Delphi 在线帮助的一些地方，常把函数和过程合称例程（routine）；但是在 IDE 的很多地方又用过程（procedures）来合称它们。本书的某些地方使用过程来合称函数和过程。如果函数和过程隶属于类或者对象，那么就应该叫做方法。在本节里，我们用过程来通称过程和函数。



3.2.1 作用域

一个 Unit (单元) 中, 声明于 interface 部分 (即 interface 和 implementation 关键字之间的过程称为全局过程。另一个单元 uses (引用) 这个单元后, 可以调用这些全局过程。

当然了, 也只能在 interface 部分作过程的声明。一个过程在没有声明的情况下, 可以在 implementation 部分直接实现, 从而成为一个局部过程。局部过程只能在本单元调用, 且调用位置必须在它的实现后面。

变量的作用域也是类似的, 如果定义在 interface 部分, 那么是全局的 (典型的如: var Form1: TForm1), 否则是局部的, 只能在本单元使用。

我们看单元 Unit1 的全部代码如下:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

  { 位置A }

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

  { 位置B }
  { 声明一个全局过程GlobalProc。一个全局过程可以在位置A、B、C三个地方声明, 效果是一样的 }
  procedure GlobalProc;

var
  Form1: TForm1;
```

```

    { 位置C }

implementation

{$R *.dfm}

{ 实现一个局部过程LocalProc，其他单元引用Unit1后是看不到这个过程的 }
procedure LocalProc;
begin
    ShowMessage('LocalProc被调用');
end;

procedure GlobalProc;
begin
    ShowMessage('调用LocalProc');
    { 必须在局部过程的实现位置以后调用。否则，由于局部过程没有声明而导致无法定位它。如
      果你将GlobalProc的实现移到LocalProc的实现之前，则会产生编译错误
      ——"LocalProc未被定义" }
    LocalProc;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    GlobalProc;
end;

end.

```

还可以在过程内部实现子过程，子过程的作用域就更小了，只能在父过程中被调用。如果一个过程有多个子过程，那么实现位置更靠后的子过程可以调用它前面的子过程。比如：

```

procedure TForm1.Button2Click(Sender: TObject);
var
    S: String;
    { 实现Button2Click的子过程ShowInfo。ShowInfo只能在Button2Click中被调用 }
    procedure ShowInfo(Info: String);
    begin
        ShowMessage(Info);
    end;
begin
    ShowMessage(Info);
end;

```



```

    end;
begin
    S := 'lxpbuaa';
    ShowInfo(S);
end;

```

3.2.2 参数传递

声明/实现一个过程使用的参数称为形式参数（简称形参），调用过程时传入的参数称为实际参数（简称实参）。

```

{ Info是形参}
procedure ShowInfo(Info: String);
begin
    ShowMessage(Info);
end;

var
    S: String;
begin
    S := 'lxpbuaa';
    {S是实参}
    ShowInfo(S);
end;

```

参数传递分两种：按值（by val）和引用（by ref）。这两种方式的本质区别是：

按值传递时，形参和实参是两个变量，它们开始时的值是相同的，即实参的数据被拷贝一份传递给了形参。所以此时，形参的改变不会影响到实参。

引用传递时，形参和实参是同一个变量，可以将它们之一看做是另一个的别名。所以此时，形参改变时，实参跟着改变。

默认情况下，参数是按值传递的，传递的是数据拷贝；如果加了 var 前缀，则成了引用传递。

我们看如下例子：

```

procedure TForm1.ByVal(I: Integer);    { 按值传递I }
begin
    ShowMessage(IntToStr(Integer(@I)));
    { 取得形参所在地址。你会发现它和实参地址是不同的，因为此时实参和形参是不同的两个变量 }
    I := I + 1;
end;

```

```

procedure TForm1.ByRef(var I: Integer); {引用传递I}
begin
    ShowMessage(IntToStr(Integer(@I)));
    { 取得形参所在地址。你会发现它和实参地址是相同的, 因为此时实参和形参是同一个变量 }
    I := I + 1;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    I: Integer;
begin
    I := 1;
    ShowMessage(IntToStr(Integer(@I)));           { 取得实参所在地址 }
    ByVal(I); { I =1 }
    ByRef(I); { I =2 }
end;

```

按值传递的参数可以指定默认值, 比如上面的 ByVal 可以是这样:

```
procedure ByVal(I: Integer = 0);
```

调用它时可以省掉有默认值的参数: ByVal。

带默认值的参数必须位于参数列表的最后, 如:

```
procedure ByVal(I: Integer = 0; B: Boolean);
```

是不行的, 应该改为:

```
procedure ByVal(B: Boolean; I: Integer = 0);
```

因为默认值必须是一个常数表达式, 所以 dynamic-array、procedural、class、class-reference 和 interface 等参数只能指定 nil 默认值; 而 record、variant、file 和 static-array 等类型的参数则根本不能指定默认值。

如果按值传递一个指针类型的参数, 情况会变得复杂而又很有意思。此时, 实际传递的是什么呢? 是实际数据的拷贝吗? 不, 是指针的拷贝, 也就是说形参和实参是两个指针, 不过这两个指针指向了相同地址。所以这时候, 形参和实参可以共享它们指向地址中的数据, 但如果改变了形参的指针指向, 实参的指针指向不能跟着改变。那么总结一下, 就是:

按值传递指针参数时, 实参和形参可以共享指针指向地址中的数据, 但是不能共享指针本身的指向。而引用传递时, 因为实参和形参是同一个变量, 因此实现完全共享。看下面的例子:

```

procedure TForm1.ByVal(Obj: TObject);
begin

```



```

Obj := Button1;
{ 改变形参指针指向, 实参的指针指向不会跟着改变, 因为它们是两个变量。如果仅仅是改变
  Obj的属性而不改变指向, 则实参的属性会跟着改变}

end;

procedure TForm1.ByRef(var Obj: TObject);
begin
  Obj := Button1;
  { 改变形参指针指向, 实参的指针指向跟着改变, 因为它们是同一个变量}
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Obj: TObject;
begin
  Obj := Self;
  {Self即Form1, 所以此时实参Obj的类名 (ClassName) 是" TForm1" }
  ByVal(Obj);           { 按值传递指针变量Obj}
  ShowMessage(Obj.ClassName); { 显示类名" TForm1" }
  ByRef(Obj);           { 引用传递指针变量Obj}
  ShowMessage(Obj.ClassName); { 显示类名" TButton1" }
end;

```

上面讲了这么多, 最根本的还是一句话: 按值传递时, 形参和实参是两个变量; 引用传递时, 形参和实参是同一个变量。抓住这句话, 就等于抓住了一切。

相信你还看到过如下格式的参数声明:

```

function CompareStr(const S1, S2: string): Integer;
function TryStrToInt(const S: string; out Value: Integer): Boolean;

```

其中使用了 `const` 和 `out` 关键字。如果你没有看到过这样的声明, 也不要紧, 它们是真实存在的。

`const` 声明的参数是按值传递的, 而且形参不能被改变。

`out` 声明的参数是引用传递的, 主要用于定义输出参数, 也就是说不需要输入值 (即实参不需要初始化), 实参传递给形参的值被忽略。

如果用 `const` 修饰指针参数, 那么只能通过形参修改指针地址里的数据而不能修改指针本身的指向。例如对于一个 `const` 对象参数, 可以修改其属性, 但是不能将它指向其他对象。例如:



```

procedure ShowInfo(const Form: TForm);
begin
    { 以下一句不能通过, 编译器提示:[Error] Unit1.pas(28): Left side cannot be
      assigned to}
    {Form := Form1;}
    { 但是通过其属性或者方法修改隶属于Form的数据}
    Form.Caption := 'lxpbuaa';
    ShowMessage(Form.Caption);
end;

```

在本小节的最后, 还不得不提及一种很特殊的参数类型: 无类型参数 (Untyped parameters)。声明时没有指定数据类型的参数称为无类型参数。因此, 从语法上讲, 无类型参数可以接收任何类型的数据。

无类型参数必须加 `const`、`out` 或 `var` 前缀; 无类型参数不能指定默认值。

如以下一些 Delphi 定义的过程都使用了无类型参数:

```

procedure SetLength(var S; NewLength: Integer);      { 参数S}
procedure Move(const Source;var Dest;Count:Integer); { 参数Source、Dest}
procedure TStream.WriteBuffer(const Buffer; Count: Longint);{ 参数Buffer}

```

所谓无类型参数可以接收任何类型的值, 只是从语法角度而言的。或者说, 理论上我们可以实现一个可以使用任何类型变量作为参数的过程, 但是实际上没有必要, 也不可能做到。

打个比方说, 我们想造一辆可以装载任何物体的汽车。因为是“任何物体”, 所以物体可能是任何形状, 于是这辆车必须没有车篷, 除了在几个车轮上铺一个足够大 (足够大就已经是个大问题了) 的平板外, 不能再有任何东西。这时候, 这个平板就可以看做是无类型的, 因为它上面可以坐人、摆一张桌子, 也可以赶一些动物上去站着或者躺着。尽管它可以承载很多种类的东西, 但是也是有限制的, 比如不能放一座山、也无法容纳 1 万头猪。

所以无类型参数的类型往往是有一定限制的。比如 `SetLength` 的参数 `S` 只能是字符串、动态数组等。

这种限制一般是在过程的实现中完成的, 在运行时检查参数值的实际类型。对于与开发环境关系紧密的参数, 限制也可以构筑在编译器里。

使用无类型参数的原因是无法在声明时使用一个统一的类型来描述运行时可能的类型, 如 `SetLength` 的参数 `S` 可以是字符串和动态数组, 而并没有一个统一的类型来代表字符串和动态数组类型, 所以干脆声明为无类型。而将类型限制放到别的地方实现 (如编译器)。例如 `SetLength` 的限制规则是写在编译器中的, 它只能作用于长字符串或者动态数组。你企图完成下面的功能时:



```
var
  I: Integer;
begin
  SetLength(I, 10);
end;
```

编译器编译时将给出错误信息：[Error] Unit1.pas(35): Incompatible types。导致编译中断。

小结

本小节的内容比较重要，重点是理解参数按值传递和引用传递的本质：按值传递时，形参和实参是两个变量；引用传递时，形参和实参是同一个变量。

3.2.3 声明指令

声明一个过程，可以使用 register、pascal、cdecl、stdcall 和 safecall 指令来指定参数传递顺序和参数内存管理方式，从而影响过程的运作。如：

```
function MyFunction(X, Y: Integer): Integer; cdecl;
```

这五个指令具有不同含义，如表 3-1 所示。

表 3-1 五个指令的不同含义

指令	参数存放位置	参数传递顺序	参数内存管理	适用地点
register	CPU 寄存器	从左到右	被调用者	默认。published 属性存取方法必须使用
pascal	栈	从左到右	被调用者	向后兼容，不再使用
cdecl	栈	从右到左	调用者	调用 C/C++ 共享库
stdcall	栈	从右到左	被调用者	API 调用，如回调函数
safecall	栈	从右到左	被调用者	API 调用，如回调函数。双接口方法必须使用

在一些源代码（包括 Delphi 自带的 VCL 源代码）中，你还可能看到 near、far、export 以及 inline、assemble 等指令，它们是为了和 16 位 Windows 系统或者早期 Pascal/Delphi 兼容，在目前的 Delphi 版本中，已经不具有任何意义，所以在新的开发中不要再使用。

3.3 类和类成员

类，简单地说，是众多不同类型数据的组合。买了一篮子菜，各种菜怎么分理、组织，是个问题。

一锅大杂烩或者每种菜做一盘，似乎都是菜鸟的吃法。面向对象就是合理调配菜的品种，组成一个个的类，放在不同的盘子里，而不是都投入一口大锅，也不是堆一厨房的盘子。

本节将帮你深入理解类及其成员的含义，同时也揭示了其中很多鲜为人知的东西。

3.3.1 类和类成员概述

类 (Class)，是一个包含字段 (Field，也称为域)、方法 (Method) 和属性 (Property) (事件 (Event) 是一种特殊的属性) 三种成员的构造体。

因为本书是讲“Delphi 精要”，所以对于面向对象理论中的类的概念，就不再使用什么“禽兽|家禽|鸡鸭鹅”之类例子来讲解了，如果大家对类的基本概念还不是很理解，那么可以参阅相关资料和书籍。

对象即类的实例，是使用构造函数 (在 Object Pascal 中是用关键字 constructors 标识的，它是一个特殊的类方法，通常是 Create) 来生成的一个内存块。销毁一个对象使用析构函数 (用关键字 destructors 标识，通常是 Destroy)。

字段，就是在对象中对应某项数据的变量。有的资料和书籍也将字段称为域，在本书中，为了便于理解，一般都称作字段。

而方法则是一些函数和过程。方法可以分为普通方法和类方法两种，分别用来操作对象和类。普通方法只有由类实例调用，而类方法可以由类或者类实例调用。

属性，实际上是一些需要特殊处理的字段的包装，它们的值可以用字段或者方法来存取。

以下是摘自 Forms 单元的一段代码，这段代码声明了一个 TCustomForm 类 (它是 TForm 的父类)：

```
{TCustomForm派生于另一个类TScrollingWinControl，它们构成父子关系}
TCustomForm = class(TScrollingWinControl)
private
    {声明一个字段FWindowState}
    FWindowState: TWindowState;
    {再声明一个字段FOnDestroy}
    FOnDestroy: TNotifyEvent;
    .....
    {声明一个方法SetWindowState}
    procedure SetWindowState(Value: TWindowState);
    .....
public
    {这是一个构造函数:Create}
    constructor Create(AOwner: TComponent); override;
    {这是一个析构函数:Destroy}
    destructor Destroy; override;
    {声明一个属性WindowState，它从字段FWindowState读取值，用方法SetWindowState
```



```

    保存值 (方法SetWindowState在内部将值保存到字段FWindowState) }
    property WindowState: TWindowState read FWindowState write
SetWindowState;
    { 声明一个特殊的属性——事件OnDestroy, 和WindowState不同, OnDestroy的存取都是
      通过字段FOnDestroy进行的 }
    property OnDestroy: TNotifyEvent read FOnDestroy write FOnDestroy
    .....
end;

```

在本小节最后, 我们渴望搞清楚类成员的可见性问题。对于类成员的其他深入知识, 我会开辟专门的小节来阐述。

类成员的可见性是对该类的使用者而言。在声明一个类时, 类可以被分为 5 个区域, 用以下 5 个关键字标识:

private, protected, public, published, automated。

所有的类成员都被放置在不同的区域里, 不同区域的类成员具有不同的可见性。如果类的定义和类的使用者在同一个单元内, 那么该类的所有成员无论位于哪个区域, 对于使用者而言都是可见的。一个类对于相同单元的其他类来说, 类似于 C++ 中的“友类”, 其所有成员都可以被访问。因此, 类成员的可见性设置只是在它们位于不同单元时, 才是有效的。这时候, 区域内成员的可见性规定如下:

(1) private 域: 总不可见。这个区域用来隐藏一些实现细节并防止使用者直接修改敏感信息, 比如容纳属性的存取字段和方法。

(2) protected: 派生类可见。这样既可以起到 private 域的作用, 也能给派生类提供较大的灵活性。该区域常被用来定义虚方法。

(3) public: 总可见。通常用来放置构造、析构函数等供使用者调用的方法。

(4) published: 总可见。而且这个区域的类成员有运行时类型信息, 该区域通常用来放置供使用者访问的属性和事件。

(5) automated: 总可见。而且该域的成员具有自动化类型信息, 用于创建自动化服务器。该关键字已经不再使用, 为向后兼容保留。

类的成员通常都是很明确指定了它所属区域的, 但并不总是这样, 凡事都是有例外的。比如我们在窗体上放置一个按钮并双击它生成 OnClick 事件过程后, 单元的源代码中对窗体类的定义就变成了下面的样子:

```

type
    TForm1 = class(TForm)
        Button1: TButton;
    procedure Button1Click(Sender: TObject);
    private

```



```

    { Private declarations }
  public
    { Public declarations }
end;

```

我们发现字段 Button1 和过程 Button1Click 并没有被明确地放到哪个可见性区域中。那么这时候它们的可见性按什么规则来确定呢？此时和编译指令 \$M 密切相关。

后面我们要讲：\$M 用来控制编译器是否给类生成运行时类型信息。所以，在 {\$M+} 状态，Button1 和 Button1Click 被隐含指定到 published 域；在 {\$M-} 状态，则到 public 域。

那么对于上面的 TForm1 来说，因为它现在处在 {\$M+} 状态，所以 Button1 和 Button1Click 实际上被隐含指定到 published 域。

要了解 VCL 定义的类的成员位于哪个域，有两种行之有效的办法：一是直接看类的源代码，这是最直接、最准确的（我们很感谢 Delphi 能附带绝大部分的源代码）；另外也可以看 Delphi 在线帮助，帮助中，类成员名称前有区域标记符号：绿方框的表示位于 published 区，黄方框的表示位于 protected 区，无标示的表示位于 public 区；如果是蓝箭头，则表示是一个只读属性。Delphi 在线帮助写得比较简练，而且似乎有越来越简练的趋势，因为在 Delphi 6/Delphi 7 的帮助中，将原来有的一些例子都去掉了！不过留下的确都是精华，很值得研究。

Delphi 中也有一些外购包，其中的类和组件的帮助做得相当糟糕，它根本没有方框、箭头之类的标识，也没有类定义的源代码，这时候就只能通过 IDE 的代码提示功能来猜测了！

3.3.2 深入认识方法

在上一小节，我们已经简单说明了类的成员之一——方法。但是，对于方法仅仅只这么一点认识是远远不够的。

在本小节里，我打算从不同角度对方法分类研究，借以深入认识方法。

(1) 从调用者角度可分为：

普通方法；

类方法。

普通方法只能被类实例（即对象）调用，而类方法不但可以被对象调用，还可以直接被类调用（比如构造函数 Create 和析构函数 Destroy）。我们看下面的例子：

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

```



```
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

  TOneObject = class
    { 声明一个类方法ClassProc。方法是在最前面加"Class"关键字 }
  class procedure ClassProc;
  { 声明一个普通方法OneProc }
  procedure OneProc;
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

class procedure TOneObject.ClassProc;
begin
  ShowMessage('ClassProc');
end;

procedure TOneObject.OneProc;
begin
  ShowMessage('OneProc');
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  OneObj: TOneObject;
begin
```

```

TOneObject.ClassProc;           {类方法可以直接被类调用}
OneObj := TOneObject.Create;    {Create本身也是个类方法}
OneObj.ClassProc;               {对象也可以调用类方法}
OneObj.OneProc;                 {普通方法只能被对象调用}
OneObj.Free;
end;

end.

```

类方法是从 C++ 的 static (静态) 函数借鉴而来的。实现一个类方法时, 要特别注意不要让它依赖于任何实例信息, 千万不要在类方法中存取字段、属性和普通方法。否则通过类而不是对象来调用它时, 将发生错误, 因为此时并没有实例信息。

在本小节接下来的内容里, 我们不再讨论类方法, 所有的方法都是指普通方法。

(2) 从调用机制上分:

静态方法。如下面的代码定义了 TOneObject 的静态方法 OneProc。

```

TOneObject = class
  procedure OneProc;
end;

```

没有修饰字的方法被默认为静态方法。和下面要讲的虚方法相比, 静态方法能够获得更快的运行速度, 因为它的地址是编译时确定、运行时映射的; 而虚方法为了实现某些更加高级、灵活、复杂的功能, 需要在运行时作一些附加处理 (比如动态寻址), 所以调用时相对要慢一些。

注意: Object Pascal 中的 static 和 C++ 中的 static 具有不同含义。C++ 中的 static 方法相当于 Object Pascal 中的类方法。

虚方法。虚方法使用关键字 virtual 或者 dynamic 声明, 如:

```

TOneObject = class
  procedure OneProc; virtual;
  function OneFun: Boolean; dynamic;
end;

```

其中 OneProc 和 OneFun 都是虚方法。虚方法可以在子类中进行覆盖, 从而增强方法的功能。覆盖一个虚方法应该使用 override 关键字。例如我们定义 TOneObject 的子类:

在子类中, 可以 (但不是必须的) 覆盖父类的虚方法, 从而实现更加复杂的控制。覆盖采用关键字 override:



```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

  {TParent声明了两个虚方法}
  TParent = class
    procedure OneProc; virtual;
    function OneFun: Boolean; dynamic;
  end;

  {TChild派生于TParent，并对父类的两个虚拟方法都做了覆盖}
  TChild = class(TParent)
    procedure OneProc; override;
    function OneFun: Boolean; override;
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

{ TParent }
```



```

procedure TParent.OneProc;
begin
    ShowMessage('TParent');
end;

function TParent.OneFun: Boolean;
begin
    Result := False;
end;

{ TChild }

procedure TChild.OneProc;
begin
    inherited; { inherited调用父类的OneProc的代码, 这句的结果是显示'TParent' }
    ShowMessage('TChild');
end;

function TChild.OneFun: Boolean;
begin
    Result := inherited OneFun; { 调用父类的OneFun代码 }
    if not Result then
        Result := True;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    Child: TChild;
begin
    Child := TChild.Create;
    Child.OneProc; { 会先显示'TParent' (父类代码实现的), 再显示'TChild' (子类实现的) }
    if Child.OneFun then { 条件语句成立 }
        ShowMessage('OneFun return true');
    Child.Free;
end;

end.

```





使用不同关键字声明的虚方法是有区别的。virtual 声明的称为虚拟方法，dynamic 声明的称为动态方法。它们在被调用时，派遣机制有所不同，具体可以参看第 5 章的“虚拟方法表和动态方法表”小节。

因为声明虚方法的目的都是供子类覆盖，所以虚方法一般应该声明在 protected 区域，当然不是绝对的。如果希望类的使用者能够调用这个虚拟方法，那么还可以声明在 public 域。

还得对 inherited 作点说明。inherited 并不仅仅局限在子类覆盖后的虚方法中调用父类中被覆盖的方法，实际上，inherited 可以使用在任何地方、调用子类可见的任何父类方法（包括 protected、public、published 等域的）。

抽象方法。它是虚方法的特例，在虚方法声明后加上 abstract 关键字构成，如：

```
TParent = class
  procedure OneProc; virtual; abstract;
  function OneFun: Boolean; dynamic; abstract;
end;
```

抽象方法和普通虚方法的区别：

a. 抽象方法只有声明，没有实现；而虚方法必须有实现部分，哪怕没有实际代码而只有 begin...end 头。

b. 抽象方法必须在子类中覆盖并实现后才用调用。因为没有实现的方法不能被分配实际地址，而调用一个没有实际地址的方法显然是荒谬的。

所以，抽象方法也可以被称为纯虚方法。

如果一个类中含有抽象方法，那么这个类就成了抽象类，如 TStrings 含有：

```
procedure Clear; virtual; abstract;
procedure Delete(Index: Integer); virtual; abstract;
```

等多个抽象方法。

抽象类是不应该直接用来创建实例的，因为一旦调用了抽象方法，将抛出地址异常，而我们很难保证不在某些地方调用抽象方法。所以，尽管实例化抽象类是被允许的，却是应该避免的。

因此，抽象类一般都是中间类，实际使用的总是覆盖实现了抽象方法的子类。比如常用的字符串列表类 TStrings，我们总是使用它的子类而不是它本身来构造实例，如：

```
var
  Strs: TStrings;
begin
  Strs := TStringList.Create;
  .....
end;
```

(3) 从用途来分：

重载方法。方法名相同，但参数个数或者类型不同的多个方法构成重载；重载的目的是得到多个同名但是功能不同的方法。重载是用关键字 `overload` 来指明的，比如：

```
TParent = class
  procedure OneProc; overload;
  function OneProc(S: String): Boolean; overload;
end;
```

上面 TParent 类的方法 OneProc 被重载。

重载方法的几个特点：

a. 可以分别是函数或者过程。因为在 Delphi 中，可以将过程看做一个没有返回值的函数，一个函数也可以当作过程调用。

b. 如果位于相同类中，都必须加上 `overload` 关键字；如果分别在父类和子类中，那么父类的方法可不加 `overload` 而子类必须加 `overload`。

c. 如果父类的方法是虚 (virtual 或者 dynamic) 的，那么在子类中重载该方法时应该加上 `reintroduce` 修饰字，否则会出现编译警告：“hides virtual method of base type”。当然只是编译时产生警告，如果你不顾它的警告，坚持不加修饰字，对程序运行结果也不会造成影响。如：

```
TParent = class
  procedure OneProc; virtual;
end;

TChild = class(TParent)
  procedure OneProc; reintroduce; overload;
end;
```

d. 在 `published` 区不能出现多个相同的重载方法。如：

```
TParent = class
  procedure OneProc; virtual;
end;

TChild = class(TParent)
published
  procedure OneProc; reintroduce; overload;
  { 和父类构成方法重载关系是可以的，因为在TChild的published区，只有一个OneProc方
```



法,而下面两行企图重载AnotherProc则是没可能的,编译器不允许在published区出现多个同名的方法}

```
procedure AnotherProc(S: String); overload;
procedure AnotherProc; overload;
end;
```

为什么编译器不允许在 published 出现多个同名的方法呢?别忘了在前面我们说过 published 区的类成员会生成运行时类型信息的,而类成员是通过名字区分的。因此,这时候编译器无法为成员 AnotherProc 生成运行时类型信息。

程序运行时,能够根据传入的参数来正确区分应该调用哪一个被重载的方法。

重载的概念对于普通的过程和函数也是适用的,实际上方法重载是从普通过程和函数的重载引申而来的。我之所以没有将重载内容放在 3.2 节,是因为方法重载更加复杂,在这里可以更加全面地来阐述它。

消息方法。消息方法的作用是截获并处理特定的一个消息。它使用的声明关键字是 message。如:

```
TCustomForm = class(TScrollingWinControl)
private
  procedure WMClose(var Message: TWMClose); message WM_CLOSE;
  .....
end;
```

TCustomForm 声明了一个消息方法 WMClose,WMClose 的作用是捕获并处理消息 WM_CLOSE。当 WM_CLOSE 消息到来时,方法 WMClose 被自动调用。关于消息方法是如何被自动调用的,可以参考“VCL 消息机制”的内容。

消息方法的规则命名:消息类名大写+‘_’+消息名(第一个字母大写,其余小写);当然这个规则不是强制的,你可以任意命名它。

消息方法参数声明格式:var Message: 消息类型。消息类型可以是基本消息类型 TMessage,也可以是特定的消息类型(一般是规则消息方法名头部加‘T’),如上面的可以是 TWMClose。特定消息类型都是 Delphi 预定义的,一般位于 Messages 单元;一些和具体控件相关的消息类型则在该控件的定义单元定义。

消息方法的实现类似下面的代码:

```
procedure TCustomForm.WMClose(var Message: TWMClose);
begin
  { 在本类中对消息作必要的处理 }
```



```

Close;
{ 然后调用父类对该消息的处理代码 }
inherited;
end;

```

事件驱动方法。它用来驱动一个事件。一般位于 `protected` 域，命名规则是用 “Do” 替代或者直接去掉事件名称的 “On”，常常被声明为 `dynamic` 虚方法。例如 `TWinControl` 类有：

```

procedure DoExit; dynamic;

procedure TWinControl.DoExit;
begin
    if Assigned(FOnExit) then
        { 驱动事件 OnExit }
        FOnExit(Self);
end;

```

以及：

```

procedure MouseDown(Button: TMouseButton; Shift: TShiftState;
    X, Y: Integer); dynamic;
procedure DragDrop(Source: TObject; X, Y: Integer); dynamic;

```

等等。

事件驱动方法往往是在消息方法中被调用的。当控件接收到一个消息后，该消息被迅速传递到对应的消息方法。在该消息方法中，就可以调用事件驱动方法去驱动相应的事件。比如：

```

procedure TWinControl.CMExit(var Message: TCMExit);
begin
    { 调用 OnExit 事件的驱动方法 DoExit 去驱动事件 OnExit }
    DoExit;
end;

```

事件驱动方法除了在消息方法中被调用，从而驱动一个事件外，还可以被程序员直接调用，从而实现事件模拟。比如：

```

Button1.Click;

```

可以模拟按钮的 `OnClick` 事件。

因此，此时是直接调用事件驱动方法去驱动事件的，所以并不需要有一个真实的消息。

TControl.Click 可以模拟鼠标点击事件，类似地，TWinControl.KeyPress 可以模拟按键事件。

小结

本小节从多个角度讨论了多种方法的用途和使用，重点是以下种类的方法：

- (1) 虚方法（含抽象方法）；
- (2) 重载方法；
- (3) 消息方法；
- (4) 事件驱动方法。

3.3.3 深入认识属性

声明一个属性的完整语法是：

```
property propertyName[indexes]: type [index integerConstant]
    specifiers;
```

更一般地，常常使用下面的简化格式来声明属性：

```
property propertyName: type read readField/GetMethod write
    writeField/SetMethod;
```

read 关键字表示读属性值，write 表示存属性值。它们后面可跟字段或者方法，属性依靠字段和方法来实现值的存取。read 和 write 是声明属性时使用的最重要的两个指令，其他指令我们放在本小节后面部分解释。下面的语句声明了属性 Caption：

```
property Caption: TCaption read FCaption write FCaption;
property Caption: TCaption read GetText write SetText;
```

通过对 read 和 write 进行不同的组合，可以控制属性的可读写性：

- (1) read/write 都存在时，属性是可读也可写的。比如：

```
property Test: String read FTest write FTest;
```

- (2) 只有 read 时，属性是只读的。如：

```
property Test: String read FTest;
```

如果只读属性是一个对象，那么可以修改该对象的字段和属性值，但是不会被 IDE 保存到 dfm 文件。“只读”只是保证你不能改变这个对象指针，比如不能指定到另一个对象。

- (3) 只有 write 时，则属性是只写的。如：

```
property Test: String write FTest;
```

绝大部分属性是可读也可写的，少部分是只读的。可读可写属性一般声明在 published 区，供组件用户在设计时存取属性值，只读属性一般声明在 public 区，供运行时使用代码取值（典型的如 TWinControl.Handle）。而只写属性在实际开发中几乎没有应用。

前面我们说了，属性的值既可以用字段来存取，也可以用方法存取。那么声明一个属性时，该选

择字段还是方法来实现存取呢？下面列出我的两条选择经验，供参考：

(1) 对于没有取值范围限制的属性，常常直接用字段来实现存取，如 Controls 单元的片断：

```
TControl = class(TComponent)
.....
private
.....
    FHint: string;
.....
published
.....
    property Hint: string read FHint write FHint;
.....
end.
```

(2) 当须要检查属性值的取值范围，或者须要执行一些附加操作时，通常采用方法来实现存取，如 TControl.Left 属性是这样声明的：

```
property Left: Integer read FLeft write SetLeft;
```

方法 SetLeft 实现如下：

```
procedure TControl.SetLeft(Value: Integer);
begin
    if Value <> FLeft then
        { 如果新设定的值和原来的值相等，就没有必要执行下面的代码了 }
        begin
            SetBounds(Value, FTop, FWidth, FHeight);
            { SetBounds 内部执行 FLeft := Value; SetBounds 主要功能是根据新的 Left、Top、
              Width、Height 属性值改变控件在界面上的显示位置 }
            Include(FScalingFlags, sfLeft);
        end;
    end;
```

到目前为止，我们对属性的基本知识已经有相当的了解。如果还要提高的话，则还要掌握下面的一些知识。回过头来，我们再审视一下声明属性时使用的完整语法：

```
property propertyName[indexes]: type [index integerConstant]
    specifiers;
```

在这段语法中，[] 中的项是可选的。对于 specifiers 部分，我们已经相当了解 read 和 write 了。所以，进一步的知识必然隐藏在 [] 和其他 specifiers 中了。



1. indexes

如果一个属性含有这个部分,那么该属性就具有数组的特点。indexes 就是存取数组元素时必须用到的参数(我们将它理解为广义的数组元素索引)。一个很典型的例子是 TCanvas 类定义的属性 Pixels:

```
TCanvas = class(TPersistent)
private
    function GetPixel(X, Y: Integer): TColor;
    procedure SetPixel(X, Y: Integer; Value: TColor);
public
    property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
end;
```

当引用属性Pixels时,X和Y的值被自动传给GetPixel和SetPixel方法,这样GetPixel和SetPixel就可以存取数组中有X和Y指定位置的元素了。

2. index integerConstant

如果声明属性时指定了这个部分,就可以让多个属性共享相同的存取方法。integerConstant 是一个整型常数,表示属性的索引。我们看如下一个例子:

```
type
    TMyComponent = class(TComponent)
private
    function GetProp(index: Integer): String;
    { 此时,存取方法必须保存索引参数,属性索引integerConstant被自动传入,从而可以
      区分当前是哪个属性调用这个方法 }
    procedure SetProp(index: Integer; Value: String);
published
    { 属性Prop1和Prop2分别被索引为0和1,它们共享存取方法GetProp和SetProp }
    property Prop1: String index 0 read GetProp write SetProp;
    property Prop2: String index 1 read GetProp write SetProp;
end;

implementation

function TMyComponent.GetProp(index: Integer): String;
begin
```



```

    case index of
      0: Result := 'Prop1';      { 取得Prop1的值 }
      1: Result := 'Prop2';      { 取得Prop2的值 }
    end;
  end;

  procedure TMyComponent.SetProp(index: Integer; Value: String);
  begin
    case index of
      0: Prop1 := Value;        { 保存Prop1的值 }
      1: Prop2 := Value;        { 保存Prop2的值 }
    end;
  end;

```

例子中，从 TComponent 派生一个类 TMyComponent。TMyComponent 声明了两个 published 类型属性 Prop1 和 Prop2。由于这两个属性的存取过程非常相似，所以让它们共享存取方法 GetProp 和 SetProp。

此时必须使用 Get 和 Set 方法，而不能使用字段方式；并且 Get/Set 方法必须包含属性索引参数 index；index 必须是 Get 方法的最后一个参数，是 Set 方法的倒数第二个参数（即位置必须在 Value 参数紧前）。

3. specifiers

这部分包含声明属性时使用的一些指令，可以是 :read/write、stored、default/noddefault和implements。read/write是基本指令，前面已经详细讲过了，这里不再重复。而其他指令则是可选的。

(1) stored。

指定该属性的值是否保存到dfm文件，后可跟True或者False，默认是True。如果指定了“stored False”，则在设计时设置的属性值不会被保存到dfm文件。

只有stored位于True状态时，我们紧接着讲述的default/noddefault指令才会起作用。

(2) default/noddefault。

指定该属性的值在何种情况下才保存到dfm文件。default后跟和属性同类型的常数，这个指令完整的意图就是：只有当用户设置的属性值和default后的常数不相等时，才保存这个值到dfm文件。

千万不要望文生义，认为default是用来指定属性的默认值。在Delphi中，指定属性的默认值需要使用以下两种方法：

人工的。

显式地给属性赋值（一般位在构造函数中，且在继承父类构造函数之后）。如：



```

TDemoButton = class(TButton)
private
    FNameEx: String;
public
    constructor Create(AOwner: TComponent); override;
published
    property NameEx: String read FNameEx;
end;

constructor TDemoButton.Create(AOwner: TComponent);
begin
    inherited;
    { 初始化属性NameEx的值为'FNameEx' }
    FNameEx := 'FNameEx';
end;

```

自动的。

编译器自动给属性指定默认值。不同类型的属性,自动指定的默认值不同。通常:数值类指定为0,指针类指定为nil,字符串类指定为",布尔类型为False等等。

如果在子类希望去掉从父类继承的属性的default指令,那么需要重新发布该属性,并加上nodefault指令。

在一个属性定义中,当没有显式指定default/nodefault时,相当于指定了nodefault指令,也就是说nodefault是默认的。

另一个值得注意的问题是在数组属性中,default指令的含义和上面所讲大相径庭。数组属性中的default意思是:该属性是对象的默认属性,可以直接使用“对象.数组属性(索引)”的方式引用属性。如:

VCL 中的类 TStrings 是这样声明的:

```

TStrings = class(TPersistent)
public
    property Strings[Index: Integer]: string read Get write Put; default;
    .....

```

那么定义:

```

var
    AStrings: TStrings;

```



则：

AStrings.Strings[I]等价于 AStrings [I]。

很显然，加 default 指令的数组属性在一个类中只能有一个，否则编译器就搞不清楚 AStrings [I] 是引用哪个数组属性了。

stored 和 default/nodetault 统称为属性的存储指令（Storage specifiers），它们只是影响属性值保存到 dfm 文件的方式以及运行时类型信息的生成和维护，不影响对象的运行过程。因此，它们只对 published 域的属性起作用。

存储指令主要是用在组件开发中。我们考虑下面一种情况。

已经开发了一个组件，并且在某些项目中已经使用了它：

```
TOneComponent = class(TComponent)
private
    FB: Boolean;
public
    constructor Create(AOwner: TComponent); override;
published
    property B: Boolean read FB write FB;
end;

implementation

constructor TOneComponent.Create(AOwner: TComponent);
begin
    inherited;
    FB := False;      { 属性B的默认值设为False }
end;
```

如果项目中使用的组件 TOneComponent 的属性 B 默认值没有被改变，那么 dfm 文件中存储 B = False。

有一天，我发现 TOneComponent 的属性 B 默认值设置为 True 会好一些，并希望打开已经开发的项目时，TOneComponent 的属性 B 默认值也能自动变为 True。这时候问题就来了，因为 dfm 文件已经存储 B = False，所以 TOneComponent 的属性 B 不可能自动变为新的默认值 True。这该怎么办呢？

利用存储指令可以解决这个问题。那就是首次开发 TOneComponent 时，就将 B 用如下格式声明：

```
property B: Boolean read FB write FB default False;
```

这样 B = False 不会被保存到 dfm 文件。现在要修改 B 的默认值，只须在构造函数 Create 中写：FB := True 即可。

(3) implements。

这个指令可以帮助属性（该属性本身必须是一个类或者接口）实现一个接口。这是一个非常高级的指令，一般开发中无须用到，所以我们也不详细研究它了。感兴趣的朋友可以参考相关的资料和 VCL 源代码。

小结

本小节消息讨论了属性的声明方法。其中重点是两类指令关键字的使用：

(1) read/write。

(2) default/nodefault。

3.3.4 深入认识事件

在前面各节中，我们对事件有了一些了解。本小节将从三个方面来深入讨论事件：

1. 事件的本质

(1) 事件是属性。

(2) 事件是方法指针。

我们首先看 Controls 单元的一段代码：

```
type
    TNotifyEvent = procedure(Sender: TObject) of object;

TControl = class(TComponent)
private
    FOnClick: TNotifyEvent;
    .....
protected
    { 事件OnClick 的声明格式印证了"事件是属性" }
    { 同时, 事件总是属于一个方法类型, 如OnClick 属于TNotifyEvent 类型, 所以说"事件
      是方法指针" }
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
    .....
end;
```

2. 事件是如何被关联的

如果我们在窗体上放置一个 TButton，双击这个按钮，IDE 自动生成方法：

```
procedure Button1Click(Sender: TObject);
```

然后我们可以在Button1Click的实现中添加事件处理代码。程序被编译时，OnClick就被关联到方法Button1Click上。

这个关联关系也会被保存到dfm文件中，在窗体上单击右键，选择“View as text”，我们看到如下的dfm文件内容：

```
object Button1: TButton
.....
Caption = 'Button1'
OnClick = Button1Click
end
```

观察上面的文件内容，我们发现事件和属性的保存并没有本质区别，都是采用“=”将属性/事件名和值连接起来。

3. 事件是如何被驱动的

在“深入认识方法”一小节讲了事件驱动方法可以驱动事件。但是并不只是事件驱动方法才可以驱动一个事件，在这里，有必要总结一下驱动事件的方法：

(1) 通过发送消息。例如：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Button1Click');
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  { 给 Button1 发送鼠标左键按下和松开消息，从而驱动 Button1 的 OnClick 事件 }
  SendMessage(Button1.Handle , WM_LBUTTONDOWN, 0, 0);
  SendMessage(Button1.Handle , WM_LBUTTONUP, 0, 0);
end;
```

(2) 调用事件驱动方法。如：

```
Button1.Click;
```

(3) 通过事件调用事件关联的处理方法。如：

```
var
  ButtonClick: TNotifyEvent;
begin
  ButtonClick := Button1.OnClick;
  if Assigned(ButtonClick) then { 首先判断该事件是否有关联的处理方法 }
    ButtonClick(Button1);
end;
```



(4) 直接调用事件关联的处理方法。如：

```
Button1Click(Button1);
```

3.3.5 类成员重新声明

如果父类的一个成员在子类中是可见的，那么可在子类中声明同名的成员。如果该过程不是覆盖和重载的，那么称为类成员的重新声明。重新声明后，从父类继承下来的该成员在子类中将不再可见。

由于字段在规范使用时，都是在类的内部使用，所以我们不再考虑字段的重新声明。

1. 方法的重新声明

构成方法的重新声明的要素是方法同名，而不关注方法类别（函数还是过程）、参数、所在区域是否改变。如：

```
type
  TParent = class
    public
      procedure DemoMethod(I: Integer);
    end;

  TChild = class(TParent)
    private
      function DemoMethod: Integer;
      { 上面一句代码已构成DemoMethod的重新声明，在TChild中已经不可能继承TParent的
        DemoMethod，虽然它是public的。 }
    end;
```

VCL 库中有一个很典型的重新声明方法的例子，那就是类的构造函数：

在 System 单元，TObject 是如下定义的：

```
TObject = class
  constructor Create;
  .....
end;
```

但是在 Classes 单元定义 TComponent 类时，重新声明了 Create，给它加上了一个参数：AOwner，表示对象的拥有者：

```
TComponent = class(TPersistent,
  IInterface, IInterfaceComponentReference)
```

```

.....
public
    constructor Create(AOwner: TComponent); virtual;
.....
.....
end;

```

一般地，几乎从来不重新声明方法，因为这样极大地降低了类架构的可读性，容易引起错误和混乱。如果在开发中确实需要实现类似功能时，总是用虚方法、抽象方法和方法覆盖、重载代之等手段来代替。

2. 属性的重新声明

如果子类的属性没有指定数据类型，那么这个过程称为属性覆盖，否则是属性重新声明。在属性覆盖和重新声明时都可以改变属性的操作指令。

在 VCL 中，常常定义一些中间类。然后从中间类派生子类，并进行必要的属性覆盖和重新声明。这个子类被作为一个最终类，提供给开发者使用。在组件开发中，这样一个过程称为子类化。

需要说明的是，和方法重新声明相似，属性也是极少重新声明的，而属性覆盖则用得很多，特别是在子类化过程中。

如 TWinControl 的 protected 域包含很多未公开属性：

```

TControl = class(TComponent)
.....
protected
    .....
    property Caption: TCaption read GetText write SetText stored
        IsCaptionStored;
    property Font: TFont read FFont write SetFont stored IsFontStored;
    property OnClick: TNotifyEvent read FOnClick write FOnClick stored
        IsOnClickStored;
    property OnMouseDown: TMouseEvent read FOnMouseDown write FOnMouseDown;
    .....
.....
end;

```

其间接子类 TButton 则使用属性覆盖方法将它们公开了：

```

TButton = class(TButtonControl)
.....
published

```



```

.....
property Caption;
property Font;
property OnClick;
property OnMouseDown;
.....
.....
end;

```

小结

本小节讲述了面向对象编程中类成员重新声明应该遵循的规则。

在实际开发中，为了避免类架构设计的混乱，保证其可读性、可维护性，几乎从来不重新声明父类和祖父类的类成员。如果有实现类似功能的需要，基本都设置作用域、采用虚方法、抽象方法、成员覆盖、方法重载等手段来完成。

3.3.6 inherited 释疑

相信大家对于 “inherited” 这个词并不陌生了。但是有些人可能对它有些误解，比如下面一种情况：当有人对你提起 “inherited” 时，你可能会马上想到 “virtual”、“dynamic” 和 “override” 等等。这就是一种误解。之所以产生这样的误解，是因为我们经常覆盖（override）一个虚方法（virtual 或者 dynamic）后，使用 inherited 来调用父类的方法。但是要记住的是，inherited 并不仅仅只用在 override 之后。

在 Object Pascal 中，“inherited” 只是一个关键字，和 virtual、dynamic、override 等并没有任何直接联系。“inherited” 的完整语法是：

```
inherited XXX(ParamsList);
```

对此说明几点：

(1) 上述语法表示调用父类的方法 XXX，其中 ParamsList 表示参数列表。在前面我们已经说过，父类中除 private 外，其他域的方法对于子类都是可见的（如果没有显式指定域，则编译指令 {\$M} 开启时，隐含为 published 域，否则为 public 域）。因此，inherited 可以调用父类任何在 private 域以外的方法。

(2) 如果直接使用 “inherited”，没有指定方法名和参数，则表示调用父类的同名方法（但当父类有多个该同名方法时——如该方法已被重载，则 inherited 必须使用完整格式，否则编译器不能确定你到底要调用这些同名方法中的哪一个）。

(3) 因为本质上，inherited 就是调用父类的某个方法，所以和调用本类的某方法相比，并没有特别的的不同。“inherited” 仅仅就是告诉编译器：我要调用的是父类而不是其他类的方法而已。所以可以在任何地方使用它，例如条件中：

```

type
  TParent = class

```

```

    procedure OneProc;
  end;

  TChild = class(TParent)
    procedure Proc;
  end;

implementation

procedure TParent.OneProc;
begin
  {Some code}
end;

procedure TChild.Proc;
begin
  if 1 = 1 then      { 我这里随便给了一个条件 "1 = 1" }
    inherited OneProc;
end;

```

3.3.7 接口的真相

接口 (interface) 在 Delphi 中是一个很有意思的东西。Delphi 3 开始支持接口, 从而形成了 COM 编程的基础; 然而, Delphi 中的接口也可用在非 COM 开发中, 实现类似抽象类 (含有抽象方法的类) 的功能, 从而弥补了 Delphi 中不能多继承 (子类有多个同级父类) 的不足。这里所讲的 interface 和一个单元中的 interface 部分是完全不同的概念, 不要混淆。

说了半天, 似乎还没有解决接口是什么的问题。接口就是一组功能的实现者和使用者之间的协议。当我看到了实现一组功能的必要性, 但是其实现方式又是不能确定的且可以有很多途径, 这时候就要定义接口。接口不关心功能的具体实现过程, 但是规定了功能需要的输入条件和输出结果。

也就是说, 接口规定了功能的定义, 但是必须由类具体实现这些功能。所以, 接口的概念类似于 C++ 的纯虚类 (Pure Virtual Class)。

本书专列一小节来讲述接口, 是为了使大家理解 Delphi 中接口的真正含义。在本书的开发实例中, 基本不涉及接口的应用。

举个例子:

```

type
  IShowString = interface(IUnknown)
    procedure ShowString(S: String);

```




```
end;
TIObject = class(TObject, IShowString)
  procedure ShowString(S: String);
end;
```

上面代码中首先定义一个接口 IShowString, 它声明了一个方法 ShowString。类 TIObject 从 TObject 继承, 同时实现了接口 IShowString。

一个类可以同时实现一个或者多个接口, 如:

```
TIObject = class(TObject, I1, I2, I3);
```

接口的等级关系和类是相似的。IUnknown 是接口的祖先, 就像类的 TObject 一样。如下的声明:

```
type
  TOneObject = class
  end;
  TOneInterface = interface
  end;
```

表示 TOneObject 从 TObject 派生, TOneInterface 从 IUnknown 派生。接口也不能同时有多个同级父接口。

在以下部分, 我希望从不同角度来展示接口的具体内容。

1. 接口和类的不同

- (1) 接口中只有方法、属性, 没有字段。所以接口中的属性存取都必须通过方法。
- (2) 接口中的方法只有声明, 没有实现。实现在类中完成。
- (3) 接口中的方法没有作用域。都是公开的, 但是在类中则放到不同作用域。
- (4) 接口中的方法没有修饰字。可以认为它们都是抽象的。
- (5) 不能创建接口实例, 要使用接口, 必须在类中实现, 通过类调用接口的方法。
- (6) 在类中必须声明和实现接口的所有方法。不像类继承中可以选择。
- (7) 接口中的方法在类中实现时, 可以加 virtual/dynamic、abstract 修饰, 从而在子类中可以实现覆盖。如:

```
type
  IShowString = interface(IUnknown)
    procedure ShowString(S: String);
  end;
  TComponent1 = class(TComponent, IShowString)
  protected
```

```

    procedure ShowString(S: String); virtual;
end;
TComponent2 = class(TComponent1)
protected
    procedure ShowString(S: String); override;
end;

```

2. 接口标示

声明接口的典型语法是：

```

ChildInterface = interface(ParentInterface)
    ['{GUID}']
    { 方法列表 }
end;

```

其中的['{GUID}'] (Globally Unique Identifier, 全球惟一标示) 称为接口标示。COM 类等可以有 GUID 标示。这样我们可以通过 GUID 得到对应的接口或者 COM 类 (实例)。接口标示不是必须的。在 IDE 环境中, 按 Ctrl+Shift+G 键可以产生 GUID, 也可以调用 API 函数 CoCreateGuid 得到。如果父接口定义了标示而它的子接口没有定义, 该标示不会继承到子接口, 此时子接口被认为没有标示。Delphi 的 SysUtils 单元提供了 GUID 和 String 之间的转换函数 StringToGUID、GUIDToString。

3. 祖先 IUnknown (System 单元)

IUnknown 是这样声明的：

```

IUnknown = IInterface;
IInterface = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID; out Obj): HRESULT; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
end;

```

它声明了三个方法, 都是在内部使用。作用是实现接口计数和接口分离。

根据上面讲的“接口和类的不同”的第 6 点, 我们知道, 任何类要想实现接口, 就必须实现上面三个方法 (当然同时实现多个接口时, 三个方法只需要一次实现)。这是不是很麻烦? 幸运的是, Delphi 内部自动实现了这三个方法, 所以：

(1) 如果你的类从 TObject、TPersistent 派生, 请分别使用 TInterfacedObject 和 TInterfacedPersistent 代替 TObject、TPersistent, 它们内部实现了这三个方法。如：

```

TIOObject = class(TInterfacedObject, IShowString)

```



(2) TComponent 则直接实现了这三个方法：

```
TComponent = class(TPersistent, IInterface,
  IInterfaceComponentReference)
protected
  { IInterface }
  function QueryInterface(const IID: TGUID; out Obj): HResult;
    virtual; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;
```

所以，从 TComponent 及其子类派生的类可以实现任何接口而不需要考虑这三个方法的实现。

4. 接口方法的调用

(1) 直接分配。如：

```
var
  IShowStr: IShowString;
begin
  { 类和它们实现的接口是兼容的 }
  IShowStr := TComponent2.Create(nil);
  IShowStr.ShowString('dd');
  { 接口引用计数方法会最终销毁接口所属的对象，所以不需要显式销毁对象。我们下面会详细
    讲述这个问题。 }
end;
```

(2) 使用 TObject.GetInterface 方法。使用这个方法时，接口必须指定了标示。定义如下：

```
function GetInterface(const IID: TGUID; out Obj): Boolean;
```

调用如：

```
var
  IShowStr: IShowString;
begin
  TComponent2.Create(nil).GetInterface(IShowString, IShowStr);
  IShowStr.ShowString('dd');
end;
```

(3) 使用 RTTI 的 as 操作符。此时接口也必须指定了标示。如：

```
begin
```

```
(TComponent2.Create(nil) as IShowString).ShowString('dd');
end;
```

(4) 如果类将接口方法声明在公开域, 可以直接用类实例调用接口方法, 这时候接口方和类本身的方法没有任何区别。

可以用下面的方法判断一个接口、对象、类是否支持某个接口:

```
function Supports(const Instance: IInterface; const IID: TGUID;
  out Intf): Boolean; overload;
function Supports(const Instance: TObject; const IID: TGUID;
  out Intf): Boolean; overload;
function Supports(const AClass: TClass; const IID: TGUID): Boolean;
  overload;
```

5. 接口引用计数

接口引用计数是由 `_AddRef` 和 `_Release` 实现的。实现这两个方法的类中会有一个整数字段 (如 `TInterfacedObject.RefCount`)。当引用一个接口时, `_AddRef` 将 `RefCount` 加 1, 引用完毕后 `_Release` 将 `RefCount` 减 1。如果 `RefCount` 减少到 0 时, `_Release` 调用接口所属对象的 `Destroy` 方法销毁对象。大家看个例子:

```
var
  Obj: TComponent2;
  IShowStr: IShowString;
begin
  Obj := TComponent2.Create(nil);
  IShowStr := Obj;
  Obj.Free;
  IShowStr.ShowString('dd');
end;
```

当最后一句执行完后, 会触发异常。因为 `IShowStr` 的方法调用完毕后, `_Release` 将 `RefCount` 减到了 0, 于是调用 `Obj.Destroy`; 但是这时候 `Obj` 已经被销毁, 因而异常。

接口引用可以自动计数, 不需要显式地销毁它; 但在一些实时性很强的程序中, 你也可以使用类似如下格式来显式销毁接口实例:

```
IShowStr := nil;
```

但并不是说我们创建的对象就可以不显式地调用 `Free`、`FreeAndNil` 等销毁了。引用计数实现的内部分其实是很复杂的, 我们应该显式地销毁动态创建的对象。

6. 方法分辨



如果一个类实现了多个接口，而这些接口中有同名方法，那么应该如何区分这些接口？

(1) 如果符合重载的原则（方法名相同，但是参数不同或者一个是过程、另一个是函数），可以用 `overload` 关键字声明成重载方法。

(2) 如果完全相同。就需要使用方法分辨子句。例如：

```
type
  IShowString1 = interface(IUnknown)
    procedure ShowString(S: String);
  end;

  IShowString2 = interface(IUnknown)
    procedure ShowString(S: String);
  end;

  TComponent1 = class(TComponent, IShowString1, IShowString2)
  protected
    { 以下两行就是方法分辨子句 }
    procedure IShowString1.ShowString = ShowString1;
    procedure IShowString2.ShowString = ShowString2;

    procedure ShowString1(S: String);
    procedure ShowString2(S: String);
  end;
```

7. 接口授权

假设 `TComponent1` 实现了接口 `IShowString`，现在 `TComponent2` 也需要实现 `IShowString`，而且实现的功能和 `TComponent1` 完全一样，那么可不可以通过很简单的办法从 `TComponent1` 引用这个功能，而不需要重新抄写代码？

Delphi 中提供了属性关键字 `implements` 来实现这个引用功能，称为代理或者授权。大概意思是这样的：

```
type
  IShowString = interface(IUnknown)
    procedure ShowString(S: String);
  end;

  TComponent1 = class(TComponent, IShowString)
    procedure ShowString(S: String);
  end;
```



```

TComponent2 = class(TComponent, IShowString)
  IShowStr: IShowString;
  { 以下这句可以代替声明 ShowString }
  property ShowStr: IShowString read IShowStr implements IShowString;
end;

```

然后通过属性 ShowStr 引用接口方法。不过引用前必须“实例化” IShowStr。例如：

```

constructor TComponent2.Create(AOwner: TComponent);
begin
  inherited;
  IShowStr := TComponent1.Create(AOwner);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Component2: TComponent2;
begin
  Component2 := TComponent2.Create(nil);
  Component2.ShowStr.ShowString('dd');
end;

```

小结

本小节讲述了 Delphi 中接口的概念、作用、实现和使用方法。引入接口的目的有两个：

- (1) 模拟类的多继承关系；
- (2) 开发 COM 程序。

在 VCL 类库中，有一些基础类（如 TComponent）和 Web 类（如 TMultiModuleWebAppServices）等使用了接口来帮助实现一些功能。在 COM 类中则大量使用，如 TInterfacedObject、TContainedObject 等。

一般在开发非 COM 程序时，因为较少须要使用多继承功能，相应地也较少使用接口，特别是从已有的 VCL 类和组件扩展用户自定义类和组件时。

3.4 编译指令

编译指令（Compiler directives）决定编译器如何编译你的单元和程序。你可以将它理解为煤气灶上的煤气开关，或者音箱的“BASS”控制旋钮。

我们经常看到的如：{\$R *.dfm}、{\$R *.res} 等解释编译指令。

在 IDE 代码编辑环境按 Ctrl+O+O 键可以看到当前项目的全部编译指令设置。

Delphi 编译指令分为三类：

- (1) 开关指令；
- (2) 参数指令；
- (3) 条件指令。

3.4.1 开关指令

开关指令 (Switch directives) 只有两种状态：+/- 或者 on/off，就像电源开关一样。因此，开关指令的使用格式有：

- (1) {开关指令简写+}，{开关指令简写-}
- (2) {开关指令全称 on}，{开关指令全称 off}

比如：

```
{ $A+ }, { $A- }
{ $ALIGN ON }, { $ALIGN OFF }
```

打开 Delphi，然后选择菜单 Project|Options|Compiler，我们看到有很多选择框，如图 3-2，这些就是开关指令，勾上好比 on，不勾好比 off。

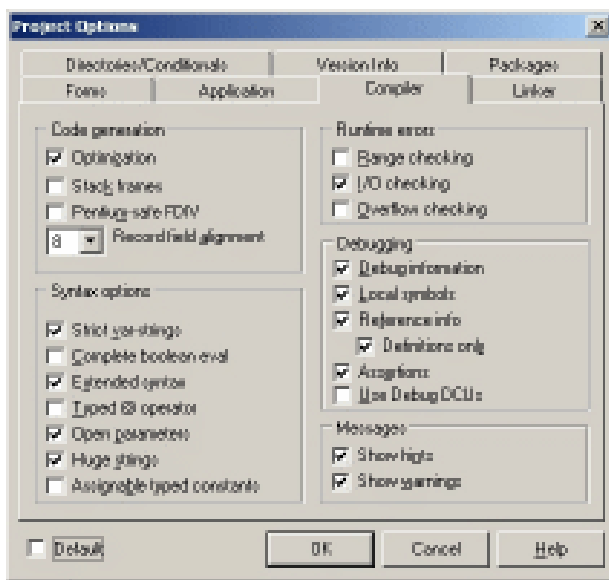


图 3-2 Compiler 页的指令

接下来，我们就用表格来分类说明它们。注意：

- (1) 类别前的 “*” 表示过时指令。
- (2) 范围的 Local 表示只对相同两个指令内的代码起作用，Global 表示作用于整个 program 或者 unit。

Code generation



类 别	对应指令	意 义	默 认	范 围
Optimization	\$O	是否优化代码。包括防止变量到 CPU 寄存器、合并分解表达式、生成中间变量等	+	Local
Stack frames	\$W	是否生成所有的过程和函数的 stack frames。一些调试工具需要此类信息	-	Local
*Pentium-safe FDIV	\$U	是否修正早期 Pentium 处理器的 FDIV 指令集浮点运算缺陷。Win95 等操作系统已经内部修正。可以在运行时查看 System 单元的变量 TestFDIV 值看是否已经修正	-	Local
Record field alignment	\$A	控制记录和类中字段的对齐方式。具体参看说明 1	+ {A8}	Local

Runtime errors

类 别	对应指令	意 义	默 认	范 围
Range checking	\$R	是否对 ShortString、有序类型和 Array 执行越界检查。打开后增加可执行文件大小、减慢速度，一般在调试阶段使用	-	Local
I/O checking	\$I	是否在 I/O 方法中自动生成调用结果检查。即：如果调用失败，是否触发异常。在{\$I-}状态，需要自行调用 IOResult 才能检查调用结果	-	Local
Overflow checking	\$Q	是否执行算术运算溢出检查。打开后增加可执行文件大小、减慢速度，一般在调试阶段使用	-	Local

Syntax options

类 别	对应指令	意 义	默 认	范 围
*Strict var-strings	\$V	当 ShortString 作为 var 参数时，是否执行严格类型检查（包括最大长度）。为向后兼容提供，对 AnsiString 无效	+	Local

续表

类 别	对应指令	意 义	默 认	范 围
Complete boolean eval	\$B	是否对布尔表达式完全运算。比如在 {\$B-} 状态, (1 = 1) or (2 = 2) 只运算 (1 = 1), 因为这时已经知道整个表达式结果	-	Local
*Extended syntax	\$X	对于函数调用和 Null 结束字符串的使用规定, 向后兼容, 已不使用	+	Global
Typed @ operator	\$T	@ (等价于 Addr) 是否得到有类型指针, 以及指向相同类型的有类型和无类型指针是否兼容	-	Global
*Open parameters	\$P	对于早期 Delphi 中代替现在的长字符串的 OpenString 的使用规定, 向后兼容, 已不使用	+	Local
Huge strings	\$H	决定 String 表示 AnsiString (+) 还是 ShortString (-)	+	Local
Assignable typed constant	\$J	是否可以修改有类型常量的值	-	Local

Debugging

类 别	对应指令	意 义	默 认	范 围
Debug information	\$D	是否生成调试信息到 dcu 文件	+	Global
Local symbols	\$L (\$D+时 有效)	是否生成本地符号(变量、常数)信息供调试器使用。{\$L+} 时, 可以在调试状态观察和修改本地符号的值; 不影响最后生成的可执行文件的大小和运行速度	+	Global
Reference info /Definitions only	\$Y (\$D+且\$L+ 时有效)	是否生成符号定义和引用位置信息表供 IDE 使用	{\$YD} (只生成 定义位置 信息表)	Global
Assertions	\$C	决定是否可以使用 Assert 全局方法	+	Local
Use Debug DCUs	无	是否允许连接含调试信息的 dcu	否	



Messages (Delphi 7 在 Project|Options|Compiler Messages 下)

类 别	对应指令	意 义	默 认	范 围
Show Hints	\$HINTS	编译时是否出现提示信息	on	Local
Show Warnings	\$WARNINGS	编译时是否出现警告信息	on	Local
WARN UNSAFE_CODE	\$WARN UNSAFE_CODE	Delphi 7 新增。编译时是否出现不安全类型转化信息	off	Local

其他重要开关指令

名 称	解 释	默 认	范 围
\$DESIGNONLY	是否生成只设计时可用包	off	Local
\$RUNONLY	是否生成只运行时可用包	off	Local
\$ObjExportAll	是否生成 obj 文件。C++ 可以使用生成的 obj 文件	off	Global
\$M	这个讲很多次了，参看“运行时类型信息”一节	—	Local

说明 1：字段对齐

字段对齐的概念适用于任何 Structured (构造) 类型，其含义在不同版本的编译器中可能不同。Delphi 6 编译器的指令 \$A 指定的对齐方式含义如下：

- { \$A1 } / { \$A- }：不对齐。
- 如果没有 packed (强制不对齐) 关键字：
- { \$A2 }：按字 (2B) 对齐。
- { \$A4 }：按双字 (4B) 对齐。
- { \$A8 } / { \$A+ }：按四字 (8B) 对齐。此为默认状态。

一般情况我们的程序都应该在默认状态 { \$A+ } 下工作。由于编译器的不同版本可能对 \$A 的设置不同，某些需要兼容不同版本编译器的程序可以设置 { \$A- } 状态，如使用 record 来读写文件、传递数据流。

3.4.2 参数指令

参数指令 (Parameter directives) 用于设定编译参数，如文件名、内存大小。格式是：

{ 参数指令 参数列表 }

比如：

```
{ $APPTYPE GUI } , { $APPTYPE CONSOLE }
```



上面这个指令决定工程是编译成图形界面还是控制台程序。

Delphi 的菜单 Project|Options|Linker 下部分是此类指令（在下面的说明中我会标明它们）。Linker 页的指令如图 3-3 所示。

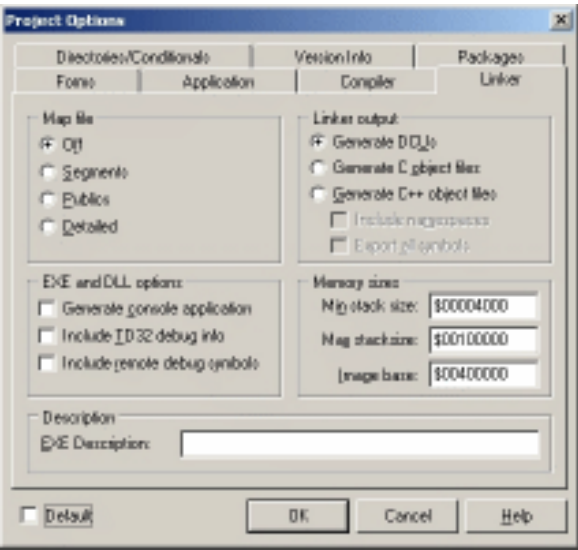


图 3-3 Linker 页的指令

Map file（连接器如何生成可执行文件的映象文件）

类 别	意 义	默 认
Off	不生成	是
Segments	包含子块列表、执行时起始地址、提示和警告信息	
Publics	Segments 及所有公开符号的列表	
Detailed	Publics 及子块自己的映射信息	

Linker output

类别	意 义	默 认
Generate DCUs	是否生成 dcu 文件	是
Generate C object files	是否生成 C 规则的 obj 文件	
Generate C++ object files	是否生成 C++规则的 obj 文件	
Include namespaces	当需要生成 C++Builder 共享的 obj 文件时，必须选择此项	
Export all symbols	当需要生成 C++Builder 共享的、包中的 obj 文件时，必须选择此项	



EXE and DLL options

类 别	意 义	默 认
Generate console application	是否生成控制台程序 对应指令{\$APPTYPE GUI}、{\$APPTYPE CONSOLE}	否 { \$APPTYPE GUI}
Include TD32 debug info	是否将调试信息写入可执行文件。可执行文件增大，但不增加运行需要内存量	否
Include remote debug symbols	是否使用远程调试器	否

Memory sizes

类 别	意 义	默 认
Min stack size	指定应用程序的栈的最小分配值 对应指令\$M	{ \$M 16384,1048576}
Max stack size	指定应用程序的栈的最大分配值 对应指令\$M \$M 指定了一个应用程序栈的大小范围，当需要值超出这个范围时，将产生 EStackOverflow（栈溢出）异常	{ \$M 16384,1048576}
Image base	指定 dll、bpl 的加载地址，不推荐用于 exe 对应指令\$IMAGEBASE	{ \$IMAGEBASE \$00400000}

Description

类 别	意 义	默 认
EXE Description	生成应用程序描述。对应指令\$D，如 {\$D 'My Application version 12.5'}	无

最后说明一下 Project Options 之 Default 选项的意思。选择 Default 表示自动将当前 Project Options 设置应用到新的工程。

其他重要参数指令

名 称	解 释	默 认	范 围
\$E	指定编译结果文件（exe、dll）的扩展名。如{\$E deu}		Global
\$IMPLICITBUILD	编译应用程序时是否自动编译引用的包中的单元。不推荐在包外使用	on	Global
\$I	在指定处插入外部文件内容。不能用在 begin 和 end 之间		Local

续表

名 称	解 释	默 认	范 围
\$L	连接 obj 文件		Local
\$Z	指定枚举值用 Byte、Word 还是 Dword 类型保存，分别对应{\$Z1}、{\$Z2}、{\$Z4}	{ \$Z1 }	Local
\$R	编译资源文件。如.rc、.res、.dcr、.dfm 等		Local

3.4.3 条件指令

条件指令 (Conditional directives) 类似于程序流程控制语句 if...then...else，它要求编译器在某个条件成立时才编译某段代码。换句话说 if...then...else 是用来控制程序执行流程的，而条件指令是用来控制编译器编译流程的。

下面一个例子演示了条件指令的使用方法：

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  { $DEFINE DEBUG } { 定义DEBUG定义 }
  { $IFDEF DEBUG } { 当DEBUG已经定义后才能编译下面的代码 }
    ShowMessage('Debug is on. ');
  { $ELSE }          { 当DEBUG没有定义时编译下面的代码 }
    ShowMessage('Debug is off. ');
  { $ENDIF }

  { $UNDEF DEBUG } { 取消DEBUG的定义 }
  { $IFDEF DEBUG } { 当DEBUG没有定义时编译下面的代码 }
    ShowMessage('Debug is off. ');
  { $ENDIF }

  { $R+ }
  { $IFOPT R+ }      { 当某编译指令在指定状态 }
    ShowMessage('range-checking编译指令打开. ');
  { $ENDIF }
end;

```

上面例子中的 `DEBUG` 是条件指令使用的标识符，标识符指定了当前环境的状态。比如 Object Pascal 预定义了以下一些标识符：

`VER140` 编译器版本。`VER140` 表示 14.0。如果是 10.0，那么会定义 `VER100`。

`MSWINDOWS` 程序运行在 `MSWINDOWS` 操作系统。

`WIN32` 程序运行在 32 位 `MSWINDOWS`。如果是 64 位 `MSWINDOWS`，会定义 `WIN64`。

`LINUX` 程序运行在 `LINUX` 操作系统。

`CPU386` 程序运行在 intel 386 或以上 CPU 上。

`CONSOLE` 是控制台程序。

`CONDITIONALEXPRESSIONS` 测试 `$IF` 指令使用。

以上这些标识符是可以在条件编译指令中直接使用的。

第 4 章 VCL 入门

VCL 是可视化组件库 (Visual Component Library) 的简称, 它是使用 Delphi 进行可视化开发的基础, 其重要性仅次于 Object Pascal。在本章里, 我们将讨论 VCL 的一些基本知识, 为下一章学习 VCL 精要作些准备。

4.1 VCL 概述

在 Delphi 出世之前, Borland 公司推出了 Turbo Pascal For Windows 来开发 Windows 应用程序, 这个软件使用的类库是 OWL (Object Windows Library)。OWL 的引入, 使 Windows 应用程序的开发工作大大简化。拿处理消息来说, 在有 OWL 之前, 必须使用一个庞大的 case 语句才能摆平, 但是有了 OWL 就不一样了, 它已经预先帮你实现了这个功能。VCL 是 OWL 的延续和发展, 都是基于面向对象理论, 不过在实现上有非常大的不同。

VCL 就是一个庞大的类、组件库, 其中类大约 1500 个, 组件约 200 多个。

在一开始, VCL 是为 Delphi 专门设计的, Object Pascal 是 VCL 使用的语言。我们知道, Delphi 差不多是世界上惟一使用 Pascal 作为开发语言的工具了, 所以 Borland 可以根据自己的需要自由地改造 Object Pascal, 从这个角度来说, Delphi 可以看做某个版本的 Pascal 编译器和实现工具。为了将 VCL 实现为一个优秀的开发类库, Delphi 的各个版本在 Pascal 中增加了许多语法特征, 这就奠定了 VCL 优异性的基础。为了共享这个优秀的类库, Borland 公司改造了 C++Builder 的编译器(它原来不过是 C/C++ 编译器大家族中的普通一员), 使它能够识别 Pascal 语法, 所以我们发现在 C++Builder 中, 居然可以直接将一个 pas 文件加入工程。实际上, C++Builder 和 Delphi 使用的组件库都是 VCL! 在 Delphi 中编写的一个组件包可以直接在 C++Builder 中使用。

VCL 采用单根架构 (Single-rooted Hierarchy), 所有类的祖先都是 TObject, 且每次只能从一个而不是多个类派生子类 (即单继承而非多继承)。Java 也使用单根架构, 而 C++不是。Object Pascal 使用单根架构的优势是:

- (1) 所有的对象都可以作为它们的父类对象来处理。比如, 所有对象都可以当作祖先类 TObject 的对象来处理。
- (2) 可在不同层次的类中加上子类共同需要的功能, 子类可以继承这些功能。比如, TObject 提供了对象构造/析构、支持运行时类型信息、支持消息处理、支持接口实现等。
- (3) 由于 TObject 统一了所有类对象的构造和析构方法, 因此, Object Pascal 强迫所有对象创建在堆 (Heap) 上, 这大大简化了对象传递的处理; 同时也借此实现了垃圾收集 (Garbage Collection) 机制。

同时，Delphi 也提供了在类中实现接口的方法来模拟多继承，基本上解决了单根架构不能实现多继承所存在的问题。

4.2 组件与控件的概念

在本章开头，我们就提到了 VCL 是可视化组件库（Visual Component Library）的简称。可视化（Visual）库（Library）很容易理解，那么什么是组件呢？

首先，一个组件就是类（Class）。该类按照面向对象编程（OOP，Object-Oriented Programming）的原理封装了一系列功能。在程序中，我们可以创建类的实例（Class Instance，即对象，Object）。

其次，运行 Delphi，你可以看到 Delphi 的组件面板，上面分很多页，如“Standard”、“Additional”、“System”等。每个组件页下有很多组件，可以拖入窗体或者数据模块进行设计。因此，我们说组件在程序设计时是可以进行可视化设计的。

综合一下，组件的概念就是：可以在程序设计时进行可视化设计的类。在 Delphi 中，一个类被注册后，就会出现在组件页，从而成为一个组件。

控件是组件的一种。这是根据组件在程序运行时的可视性来划分的，在程序运行时可见的组件就是控件。

在 VCL 中，组件可以简单地分为以下四类：

(1) 不可视组件。程序运行时是不可见的。

(2) 标准控件（即 Standard 和 Win32 组件页的组件）。用来完成 Windows 的标准图形界面，大多数开发环境（如 C++Builder、VC、VB、VS.NET 等）都提供这类控件。

(3) 图形控件。这类控件用来显示图形。

(4) 其他控件。是从现有组件扩展而来的，如 TBitBtn 等。

在 VCL 提供的类、组件和控件基础上，我们可以按照面向对象原理扩展出自己的类、组件和控件。

4.3 使用 VCL

使用 VCL，就是使用它提供的类、组件。可以通过两种途径使用 VCL 的类、组件。

(1) 设计时加入组件进行属性和事件设计，然后就可以运行程序了。这可以看做是通过静态方式使用组件。

(2) 用代码创建类的实例（对象），然后使用它提供的功能，最后销毁它。这可以看做是通过动态方式使用组件。例如：

```
unit Unit1;  
  
interface
```




```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    Button: TButton;
    procedure ButtonOnClick(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  Button := TButton.Create(Self); { 创建类 TButton 的实例 Button }
  with Button do { 设计它的属性和事件 }
  begin
    Name := 'lxpbuaa';
    Parent := Self;
    Left := 50;
    Top := 50;
    Caption := '请点击我';
    OnClick := ButtonOnClick; { ButtonOnClick 过程处理 Button 的点击事件 }
  end;
end;

procedure TForm1.ButtonOnClick(Sender: TObject);
begin
  ShowMessage('我的名字是：' + TComponent(Sender).Name);
end;
```

```

procedure TForm1.FormDestroy(Sender: TObject);
begin
    Button.Free;
    { 窗体销毁时，销毁动态创建的对象 Button；不过在这里不是必须的，因为 Button 的拥有者
      者为 Self 即 Form1。组件被销毁时，会自动销毁它拥有的所有组件 }
end;

end.

```

4.4 扩展 VCL

当现有的类和组件无法满足我们的需要时，可以从 VCL 的现有类和组件上派生新的子类和子组件。

你或许觉得自己还没有开发过类和组件，但是我告诉你，你每天都在扩展 VCL！是不是很纳闷？

运行 Delphi，选择菜单 File|New|Application，应用程序向导自动生成一个包含一个窗体 (TForm1) 的工程。现在看该窗体的源代码：

```

.....
type
    TForm1 = class(TForm) {*}
    private
        { Private declarations }
    public
        { Public declarations }
    end;
.....

```

看*这句，难道你不认为类 TForm1 是对类 TForm 的扩展么，你每天都在做这样的工作吧！在本节，我不打算将如何扩展 VCL 的话题展开，如果想了解详细的类和组件扩展知识，可阅读第 6 章。

第 5 章 VCL 精要

VCL 好像一个标准的菜谱，其中包含很多非常经典的类和组件，就是这些类和组件共同组成了 VCL 的优美架构。Delphi 产品中附带了 VCL 的绝大部分源代码（少部分构筑在编译器中间，没有公开），这大大方便了我們通过阅读源代码来了解 VCL 的原理和架构，并学习开发 VCL 组件。

VCL 库可以说是广大 Delphi 爱好者的《食经》，念不好这本经，我们的开发能力就难以攀升，老是停留在拖放控件、写点 OnClick 事件代码的菜鸟阶段，的确让自己非常尴尬。我们不甘心只能喝 VCL 的汤，还要吃它的肉，啃骨头！

本章讲述了 VCL 骨架类及 VCL 处理消息的整体机制和详细方法，从而使你从整体上把握 VCL 体系，为下一章“组件开发实战”打下坚实基础。

5.1 揭开 VCL 的神秘面纱

对于很多 Delphi 初学者和爱好者来说，VCL 很神秘、很复杂，只知道一些残缺不全、支离破碎的东西。对于如何从整体上把握 VCL，却一直找不到突破口。

本节全面分析 VCL 骨架及其重要类，并揭示类实例的生成原理（虚拟方法表和动态方法表）和运行时管理（运行时类型信息），从而确保你全面认识 VCL 的基本架构，在 Delphi 学习和使用水平上达到一个前所未有的高度。

5.1.1 VCL 架构

VCL 是 Delphi 在 Windows 平台开发的核心和灵魂。VCL 是完全面向对象、面向继承的。图 5-1 描述了 VCL 的整体架构。

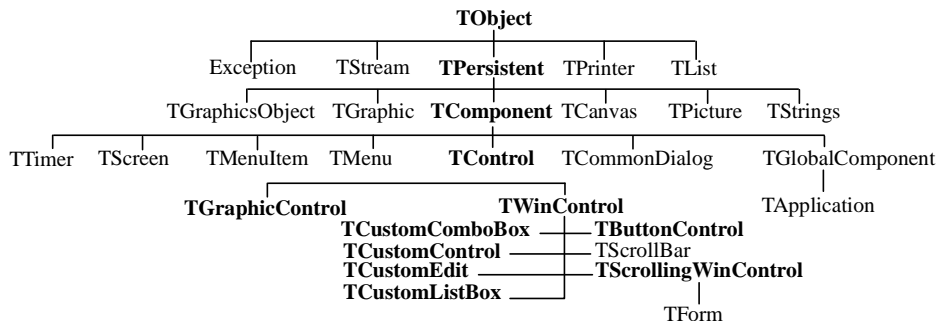


图 5-1 VCL 架构图

图中黑体构成 VCL 的主要架构。下面给大家一一介绍。

1. TObject (定义它的单元是 System, 下同)

VCL 所有类的祖先, 所有 VCL 类都直接或者间接继承于它。如果你定义类没有指明父类, 则 Delphi 自动指定 TObject 为其父类。如:

```
type
  TOneObject = class      { 等价于 TOneObject = class(TObject) }
    procedure OneProc;
  end;
```

TObject 主要定义了四类功能的 (虚) 方法:

- (1) 对象构造函数和析构函数 (Create 和 Destroy)。
- (2) 返回运行时类型信息。TObject 虽然能够提供此功能, 但是将它隐藏了 (因为是在 {\$M-} 状态编译的), 而在其子类 TPersistent 中公开。如:

```
class function ClassName: ShortString; { 返回类或者对象的类型名 }
function ClassType: TClass;           { 返回对象的类型 (即类引用) }
class function ClassParent: TClass;   { 返回类或者对象的父类类型 }
class function ClassInfo: Pointer;    { 返回类或者对象的运行时类型信息表的地址 }
```

- (3) 支持消息处理。由方法 Dispatch 和 DefaultHandler 提供。
- (4) 支持接口实现。由方法 GetInterface 和类方法 GetInterfaceEntry、GetInterfaceTable 提供。

2. TPersistent (Classes, 抽象类)

TPersistent 主要有两类功能:

- (1) 对象相互复制。AssignTo 和 Assign 这两个虚拟方法提供, 它们都要由子类具体实现。
- (2) 在流里读写属性的能力。

TPersistent 是抽象类, 不要直接创建其实例。

在 Classes 单元的类型声明代码为:

```
TPersistent = class(TObject)
```

前后你可以看到 {\$M+}、{\$M-} 字样。\$M 是一个编译指令。前面讲了 TObject 已经有提供 RTTI 的能力, 但是隐藏了, {\$M+} 指示编译器在 TPersistent 中公开 RTTI。

Delphi 规定, 打开 \$M 指令的类及其子类实例具有 RTTI, 那么这样一来, TPersistent 及其子类的实例都具有 RTTI。

3. TComponent (Classes, 抽象类)

TComponent 具有四类主要功能:

- (1) 注册后可以出现在组件页；设计时可见、可以管理；运行时不可见。
- (2) 可以拥有别的对象而成为其他对象的拥有者 (Owner)。
- (3) 加强了流读写能力。
- (4) 可以转化为 ActiveX 控件和别的 COM 类。

TComponent 是抽象类，不要直接创建其实例。如果你需要开发运行时不可见组件，可以从 TComponent 继承，否则可以从 TWinControl 或其子类继承。

4. TControl (Controls)

TControl 是控件类。所谓控件，是运行时可见的组件。VCL 所有控件都是 TControl 的直接或间接子类。

5. TWinControl (Controls)

TWinControl 是所有窗口类控件的祖先类。窗口控件有以下特点：

- (1) 可以有输入焦点。
- (2) 可以接收键盘输入。
- (3) 可以作为其他控件的容器。
- (4) 有句柄 (Handle) 属性。

TWinControl 定义了窗口控件的共同属性、方法、事件。对于不同类型的窗口控件，VCL 定义了 TCustomComboBox、TCustomEdit 等多个子类。大部分窗口控件都不是直接从 TWinControl 而是从其子类派生。TWinControl 的窗口图像是通过调用 Windows 底层方法内部完成的，而不能自定义绘制图形图像。

6. TGraphicControl (Controls)

TGraphicControl 是所有非窗口类控件的祖先类。

非窗口控件也有四个特点，且这些特点和 TWinControl 的四个特点正好相反，所以它是轻量级控件，资源消耗比 TWinControl 少很多。

TGraphicControl 增加了非常重要的 Canvas (画布，TCanvas 类型) 属性，从而提供在控件表面自定义绘制图形图像的能力；增加了 Paint 方法来响应 WM_PAINT 消息，实现自定义绘制。如在 TSpeedButton 上可以绘制一个图像。

特别指出：TCustomControl 从 TWinControl 继承，是窗口类。但是也具有非窗口类的特点：具有 Canvas 属性和 Paint 方法。

所以 TCustomControl 对象因为具有可见窗口，可以和用户交互；同时还因为具有 Canvas 属性和 Paint 方法，也能完成表面自定义绘制，如可以绘图。比如 TPanel 是 TCustomControl 的间接子类，如果公开从 TCustomControl 继承的属性 Canvas，那么就可以利用 Canvas 来绘图。

小结

本小节提供了 VCL 骨架类图，并详细讲述了 VCL 骨架类 TObject、TPersistent、TComponent、

TControl、TWinControl 和 TGraphicControl 以及 TCustomControl 所具有的特性和提供的能力。

通过学习这些特性和能力,我们发现 VCL 是一个层次分明、结构完善的优美体系。准确掌握本节所讲各骨架类的特性和能力,是从整体上把握 VCL 体系结构的关键。

某一天,你拿了一份别人写的代码来看,你看到此人定义了很多类。如果你可以在本节提供的 VCL 架构图中为这些类找到正确的位置,而不是恰恰相反——感到迷惑、找不到北,那就是值得高兴的一件事情了。

5.1.2 构造和析构的内幕

在 System 单元可以看到(查看 System 单元的技巧是:在 uses 部分人工写入 System,然后按住 Ctrl 在 System 上单击;但是编译程序时应该去掉这个单元,因为它被编译器自动 uses 的)祖先类 TObject 的构造函数和析构函数的声明和实现代码如下:

```
type
  TObject = class
    constructor Create;
    .....
    destructor Destroy; virtual;
  end;

constructor TObject.Create;
begin
end;

destructor TObject.Destroy;
begin
end;
```

我们发现 Create 和 Destroy 的实现代码一行也没有。那么是拿什么来构造和析构对象的?实际上,并不是真正用 Create 和 Destroy 来构造和析构对象。揭开现象看本质,真正实现构造和析构的是全局过程 _ClassCreate 和 _ClassDestroy。当程序员调用 Create 和 Destroy 来构造和析构对象时,编译器会自动在它们之前插入 _ClassCreate 和 _ClassDestroy 的代码。也就是说,Create 和 Destroy 只是在对象已经构造后和析构前初始化和反初始化对象成员。

因此,在开发 VCL 组件时,常常有类似如下的代码:

```
type
  TDemoClass = class(TComponent)
  private
    OneObject: TOneClass;
```





```

.....
public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
.....
end;

.....

constructor TDemoClass.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    { 或者直接写" inherited;"。当要继承的一个方法已被重载时，必须书写完整，不然编译器
      不知道你要继承哪个方法。这行继承父类的 Create 代码}
    OneObject := TOneClass.Create(Self);
    { 初始化本类的一些成员，或者给它们动态分配内存}
end;

destructor TDemoClass.Destroy;
begin
    OneObject.Free;
    { 或者 FreeAndNil(OneObject); 人工销毁本类的一些成员或者释放它们占用的内存}
    inherited Destroy;
    { 继承父类的 Destroy 代码}
end;

```

你可能还有一个疑问，就是 TObject.Create 并没有参数，这儿怎么跑出一个 AOwner(拥有者)来？是因为 TComponent 已经重新声明了 Create 方法：

```

constructor Create(AOwner: TComponent); virtual;

```

在 TComponent 及其子类中，已经不再可能使用无参数的 TObject.Create 了。

另外，在整个构造和析构过程中还调用了 TObject 两个虚方法：AfterConstruction 和 BeforeDestruction。在子类中覆盖它们后，可以作一些附加工作。比如：在 TCustomForm (TForm 的父类) 中用来触发事件 OnCreate 和 OnDestroy：

```

type
    TCustomForm = class(TScrollingWinControl)
    .....
public

```

综上所述，一个对象总的构造和析构过程如下：

我们可以使用如下的代码验证上面的过程：

```
unit Unit1;  
  
interface  
  
uses  
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,  
    Forms, Dialogs;  
  
type  
    TForm1 = class(TForm)  
        procedure FormCreate(Sender: TObject);
```




```
    procedure FormDestroy(Sender: TObject);
  private
    C: Integer;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
  end;

var
  Form1: TForm1;
```

implementation

```
{ $R *.dfm }
```

```
procedure TForm1.AfterConstruction;
begin
  inherited;
  Inc(C);
end;
```

```
procedure TForm1.BeforeDestruction;
begin
  inherited;
  Dec(C);
end;
```

```
constructor TForm1.Create(AOwner: TComponent);
begin
  inherited;
  C := 1;
end;
```

```
destructor TForm1.Destroy;
begin
  Dec(C);
  inherited;
end;
```

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    Inc(C);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    Dec(C);
end;

end.

```

在上面的每行可执行代码前加上断点运行程序，你可以清楚地看到它们的执行顺序。

小结

本小节揭示了构造函数 Create 和析构函数 Destroy 的真实作用，介绍了对象构造和析构的内部过程。

5.1.3 虚拟方法表和动态方法表

我们知道，TObject 是所有类的基本类，也就是说，所有类都直接或者间接派生于 TObject。尽管在 Delphi 中可以定义不从 TObject 派生的类，但是这样的方式已经很少使用了，我们不再讨论它。因此，TObject 是非常重要的。它是 VCL 的命根子。

我们知道，一个 TObject 的实例：Object（对象），实际上是一个 4 字节的指针，该指针指向对象的实际数据区（Object Data）。那么这个“对象数据区”到底是怎么回事呢？对象的字段、方法、属性、事件这些对象数据在“对象数据区”中究竟怎么组织、如何存取呢？

这的确是设计和实现 VCL 架构的一个根本性问题，是命根子的命根子。

原来，这个对象数据区是划分为很多个小区域的。这些区域分为两个部分：

- （1）头 4 个字节存放一个指针，该指针指向另一个地址区域。
- （2）其余小区域分别存储对象的各种数据成员（即字段，不包括方法。方法的入口地址被定义在另一个内存表里，虚拟方法表中有一个指针指向该表）。

头 4 个字节的指针指向的另一个地址区域是干嘛的呢？它就是我们常常听人说起的大名鼎鼎的“虚拟方法表（Virtual Method Table，VMT）”！虚拟方法表又被划分为很多个大小为 4 字节的小区域，每个区域存放一个指针，每个指针对应一个虚拟方法的入口地址。

接下来的众多小区域用来存放字段、属性值和所有非虚方法（注意这里用的是“非虚方法”而不是“非虚拟方法”，本节后面要详细讲述其中来由）的入口地址。

由以上可见，非虚方法成员的存取是相对简单的，而虚方法的寻址和调用则相对复杂得多。所以，在本节中，我们重点要搞清楚的是虚拟方法表——VMT。

1. VMT 的结构图

仔细观察图 5-2 可以看到：一个对象（Object）指针指向一个对象数据区（Object Data）；对象数据区的头 4 个字节存放了一个指针，该指针再指向虚拟方法表（VMT）。

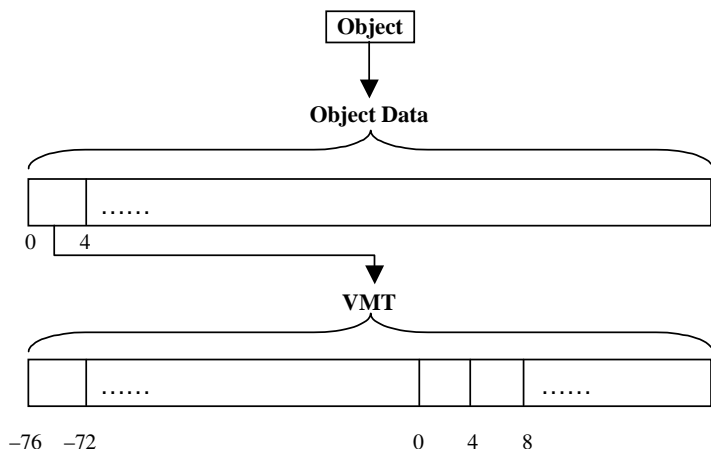


图 5-2 VMT 结构图

一个虚拟方法表从指针所指地址的负偏移-76 处开始，长度动态分配（由虚拟方法的个数确定）。虚拟方法表被分为很多小段，每段占据 4 个字节，也就是众多指针。每个指针指向一个虚拟方法的入口地址。

很显然，有了一个对象指针，按图索骥，是一定可能寻址到该对象一个指定的虚拟方法，从而调用执行这个虚拟方法。

上面的分析很对人胃口，一下子就让人明白了 VMT 是个什么东西。不过要彻底弄清 VMT，它却又露出另外一张面孔，可见，实际上并不是那么简单的。所以我还想说的是：

（1）VMT 还可以细分为两个区域，即基础信息区和用户定义虚拟方法区。

（2）VMT 对应于类而不是对象。

VMT 的负偏移区（-76~0）是基础信息区，存储了基础性数据（如实例大小）、基础性数据（如接口表、运行时类型信息表、字段表、方法表、类名和父类虚拟方法表等）的指针和所有基础性虚拟方法（这些虚拟方法都是在类 TObject 中定义的，如 AfterConstruction、BeforeDestruction、DefaultHandler、Destroy 等）的指针，所以基础信息区并不全是指针列表。这个区域所存放的数据和指针主要是用来帮助实现对象的构造和析构、运行时类型信息存取、字段和方法解析等。基础信息区的大小是固定的。

正偏移区（从 0 开始）是用户定义的虚拟方法（即所有非 TObject 定义的虚拟方法）所在区域，每 4 个字节存储一个用户定义的虚拟方法指针。这些虚拟方法不光是在本类定义的，还包括从 TObject 一直到本类的所有中间类定义的所有虚拟方法。因此，这个区的大小是根据虚拟方法的个数决

定的，不像基础信息区有固定的值。

不同的类总是具有独立的 VMT，即使这些类有继承关系或者非常近似。也就是说，VMT 是根据类的定义来生成的，跟类的实例没有关系。TComponent 和 TControl 因为是两个不同的类，因此，它们的 VMT 也是毫不相关的。

在前面章节中我们提到了类引用类型（Class reference），一个对象的类引用可以通过调用 System 单元实现的 TObject.ClassType 方法来获得：

```
function TObject.ClassType: TClass;
begin
  Pointer(Result) := PPointer(Self)^;
end;
```

可见，类引用实际上就是指向 VMT 的指针。也就是说，类引用和 VMT 有惟一对应关系。

Delphi 中的 VMT 和 C++、COM 中的 VMT 是兼容的，具有类似的结构。在将来版本的 Delphi 中，VMT 的结构可能会作一定的调整，所以一般不要直接通过地址操作 VMT，而代之以类和对象的方法、属性来存取。

2. VMT 的产生

VMT 是由编译器为程序中每个要用到的类自动生成的。VMT 对应相应的类而不是类实例。即是说，在没有创建一个类的任何实例时，该类的 VMT 已经由编译器生成了，就等着一些指针去指向它。

如果一个类 TParent 定义了一个虚拟方法 F，并且实现了部分功能；然后有一个类 TChild 派生于 TParent，并覆盖了虚拟方法 F，做了附加功能实现。这时候编译器如何编译这个方法 TChild.F，并添加到 VMT 中间呢？是不是要编译出两个方法即 TParent.F 和 TChild.F，生成两个指针加入 VMT 呢？不是的，编译器会将两个方法合成一个 F，最终只产生一个指针并加入 VMT。也就是说，TParent 和 TChild 各自的 VMT 都有 F 的指针，互不相关。

在调用构造函数创建一个对象时，对象数据区（Object Data）的头 4 个字节被分配给一个指针，然后，该指针被定向到在构造对象前就已经生成的 VMT 上。从而赋予对象调用虚拟方法的能力。

3. virtual 方法和 dynamic 方法的区别

在“类和类成员”一节中，我们讲了两类虚方法：虚拟方法（virtual）和动态方法（dynamic），它们在功能上没有区别，都是为了实现子类覆盖。区别是在调用流程上。

前面说了“非虚方法”和“非虚拟方法”是有区别的，原因就是其中还夹了个动态方法的问题。

在上面的内容中，我们对动态方法的管理和调用只字未提，只是为了避免同时端出太多话题，从而偏离主题，影响我们对 VMT 的总体理解。不是有所谓“话分多头，各表一枝”嘛！

从上述 VMT 的知识可以知道，virtual 方法被全部列入了 VMT 的正偏移区，当一个对象请求调用 virtual 方法时，可以在类的 VMT 中直接寻址，然后马上调用。但是调用一个 dynamic 方法就没有这么便宜了。

动态方法和 VMT 负偏移的基础信息数据区有关系，请看下面：

在 VMT 负偏移-48 处，是这么定义的：

-48 Pointer pointer to dynamic method table (or nil) (指针指向动态方法表 (DMT))

可见，对于一个类，它用另外一个 DMT (即动态方法表) 来存储动态方法的入口地址。这是虚拟方法和动态方法寻址和调用的一个重大区别。而 DMT 又是依赖于 VMT 来实现的。

DMT 是一系列的指针列表，和 VMT 的正偏移区类似，存放了本类定义的和从父类继承并覆盖后的动态方法入口地址。注意，如果 TChild 并没有覆盖 TParent.F，那么 F 是不会存放到 DMT 的。

因而，虚拟方法和动态方法相比，使用动态方法可以节省内存，因为它不存放未曾覆盖的虚方法指针。而且调用一个虚拟方法和调用在 DMT 中存放了入口地址的动态方法相比，速度也没有显著的差异 (不过是多了一个 DMT 寻址而已)。而如果使用虚拟方法，即使 TChild 并没有覆盖它，TParent 和 TChild 的 VMT 也都存放有 F 的入口地址。

但是如果要调用 DMT 没有入口地址的动态方法时，就麻烦了，这时候需要到父类甚至祖父类的 DMT 去寻址，才能最终完成一次调用，所以速度大打折扣。

可见，虚拟方法追求的是速度，而动态方法节省的是空间。一个是以空间换取时间，一个是以时间换取空间，所谓有得必有失，而处理好了，有所失自然也有所得。国外的一句俗语“上帝关上一道门，就会打开另一扇窗”，讲的也就是这个意思。所以我们在生活、工作、学习中遇到挫折时，没有必要悲伤、气馁；失去了一些东西，也很可能已经获得了另一些东西……哦，扯远了。☺

Delphi 之所以提供 dynamic 这样一种虚方法实现机制，是为了在下面一种情况下实现 VMT 内存节省：一个父类声明了大量虚方法，并需要派生大量子类但子类很少覆盖父类虚方法时。这时候，如果采用 virtual，那么每个子类的 VMT 都包含这个方法指针，显然在内存开销上划不来；而如果采用 dynamic 则节省多了。

注意：DMT 机制是 Object Pascal 特有的性质，因此和 C++、COM 中的 VMT 并不兼容，所以开发跨语言使用的功能时则应该采用 virtual 实现虚方法，牺牲一点空间。

小结

本小节详细讲述了对象成员的存取载体——虚拟方法表和动态方法表，并对二者作了详细对比，指出各自的优势、劣势和应用范围。

理解本小节所讲内容，有利于深入认识对象的构造原理和运作机制，有利于深入理解面向对象的概念，有利于深入领悟指针的强大作用。

5.1.4 TObject 如何使用虚拟方法表

上一小节我们已经详细分析了虚拟方法表的内部结构，但是没有提到类和对象使用虚拟方法表的具体方法。

在本小节里，我们深入分析作为万类之源的 TObject 如何存取虚拟方法表，以便加深对虚拟方法

表的认识。

TObject 类被编译时，编译器构筑一个 TObject 虚拟方法表。虚拟方法表的基础信息区存放了大量基础数据和指针，具体分布如下所示：

偏移量/Byte	数据类型	内容
-76	Pointer	虚拟方法表本身地址
-72	Pointer	接口方法表地址
-68	Pointer	自动化信息表地址
-64	Pointer	类实例初始化信息表地址
-60	Pointer	运行时类型信息表地址
-56	Pointer	字段定义表地址
-52	Pointer	方法定义表地址
-48	Pointer	动态方法表地址
-44	Pointer	类名（ShortString 类型）地址
-40	Cardinal	类实例大小
-36	Pointer	父类虚拟方法表地址
-32	Pointer	虚拟方法 TObject.SafeCallException 入口
-28	Pointer	虚拟方法 TObject.AfterConstruction 入口
-24	Pointer	虚拟方法 TObject.BeforeDestruction 入口
-20	Pointer	虚拟方法 TObject.Dispatch 入口
-16	Pointer	虚拟方法 TObject.DefaultHandler 入口
-12	Pointer	虚拟方法 TObject.NewInstance 入口
-8	Pointer	虚拟方法 TObject.FreeInstance 入口
-4	Pointer	虚拟方法 TObject.Destroy 入口

我不能凭空捏造虚拟方法表的使用方法，所以还是打开 System.pas 单元，用 Delphi 的源代码说话。System 定义了一些常数来对应上面基础数据区的各项偏移：

```

{ Virtual method table entries }

vmtSelfPtr           = -76;
vmtIntfTable         = -72;
vmtAutoTable         = -68;
vmtInitTable         = -64;
vmtTypeInfo          = -60;
vmtFieldTable        = -56;
vmtMethodTable       = -52;
vmtDynamicTable      = -48;
vmtClassName         = -44;
vmtInstanceSize      = -40;
vmtParent            = -36;
    
```

5.1
揭开 VCL 的神秘面纱

```
vmtSafeCallException      = -32;  
vmtAfterConstruction      = -28;  
vmtBeforeDestruction      = -24;  
vmtDispatch               = -20;  
vmtDefaultHandler         = -16;  
vmtNewInstance            = -12;  
vmtFreeInstance           = -8;  
vmtDestroy                = -4;
```

从上述各个常数的命名可以知道，它们分别对应于虚拟方法表基础信息区的各项数据和指针的相对偏移。

相应地，TObject 定义大量方法来操作基础信息区的数据和指针对应方法。下面我们摘录一些帮助说明基础信息的操作方法：

```
TObject = class  
.....  
  function ClassType: TClass;  
  class function ClassName: ShortString;  
  class function ClassParent: TClass;  
  class function ClassInfo: Pointer;  
  class function InstanceSize: Longint;  
  class function GetInterfaceTable: PInterfaceTable;  
.....  
end;
```

这些方法的实现如下（在下面的代码中大量使用了 Self 变量。注意：在一个普通方法中，Self 代表类实例；在一个类方法中，Self 代表类本身，即类引用。而且 Self 只能在方法中引用，在普通函数和过程使用 Self 是不可能的）：

```
{ 取得实例的类型，即类引用}  
function TObject.ClassType: TClass;  
begin  
  Pointer(Result) := PPointer(Self)^;  
{ 从 Pointer(Result) 我们知道返回值，即类引用实际上是一个指针。}  
{ 那么这个指针指向在哪里呢？这个函数中的 Self 是类实例而不是类，所以我们从  
  PPointer(Self)^可以知道，这个指针指向了虚拟方法表的 0 偏移处。也就是说，类引用  
  实际是指向虚拟方法表的指针。}  
end;
```

```

{ 取得类名 }
class function TObject.ClassName: ShortString;
begin
    Result := PShortString(PPointer(Integer(Self) + vmtClassName)^)^;
    { 这里的 Self 是类而不是类实例, 因此, Integer(Self) 是虚拟方法表的地址。然后加上类名
      指针在 VMT 中的偏移量 vmtClassName, 再转化为 PShortString 类型的指针, 最终取得
      一个 ShortString 类型的字符串, 即类名 }
end;

{ 取得父类引用 }
class function TObject.ClassParent: TClass;
begin
    Pointer(Result) := PPointer(Integer(Self) + vmtParent)^;
    if Result <> nil then
        Pointer(Result) := PPointer(Result)^;
    { 最终得到的其实是一个指向父类的虚拟方法表的指针。 }
end;

{ 取得运行时类型信息 }
class function TObject.ClassInfo: Pointer;
begin
    Result := PPointer(Integer(Self) + vmtTypeInfo)^;
end;

{ 取得类实例大小 }
class function TObject.InstanceSize: Longint;
begin
    Result := PInteger(Integer(Self) + vmtInstanceSize)^;
end;

{ 取得接口表 }
class function TObject.GetInterfaceTable: PInterfaceTable;
begin
    Result := PPointer(Integer(Self) + vmtIntfTable)^;
end;

```

小结

本小节在上一小节“虚拟方法表和动态方法表”的基础之上, 通过引用 System.pas 单元的源代码, 具体解析了类和类实例对虚拟方法表基本信息区的数据和指针的操作方法。

讲述上述内容的目的是帮助我们深入学习类和对象运作的基本原理。在实际开发中, 我们不提倡



(当然 Borland 公司也是这样的态度) 使用上述方法直接操作虚拟方法表。

5.1.5 运行时类型信息

运行时类型信息的英文名是: runtime type information, 简称为 RTTI。在 Delphi 中, RTTI 是程序设计时和运行时存取类和对象的成员的重要手段。

RTTI 主要被 Delphi 和 VCL 在内部使用, 比如实现 “Object Inspector”, 一般情况下不需要在应用程序中直接使用, 但是有时候使用是方便而且必须的。

大多数时候, 我们用 RTTI 来实现属性值的存取。但是要注意, 我们常常只能通过 RTTI 读取和设置 published 域的属性。所以本节所讲的过程和函数都只适用于 published 域的属性的存取。

有两种方法可以获得 RTTI:

(1) 调用类方法:

```
class function TObject.ClassInfo: Pointer;
```

因为它是类方法, 所以可以由类调用, 也可以由类实例调用。

(2) 调用全局函数:

```
function TypeInfo(TypeIdent): Pointer;
```

其中 TypeIdent 是一个类型标识符, 如 TButton、TColor。

Delphi\Source\VCL\TypeInfo.pas 包含了对 RTTI 的具体使用方法, 应用时必须 uses TypeInfo。下面我们列出一些常用的存取 RTTI 的函数和过程:

(1) 判断对象或者类是否存在某个属性:

使用函数 GetPropInfo:

```
function GetPropInfo(Instance: TObject; const PropName: string;  
  AKinds: TTypeKinds = []): PPropInfo; overload;  
function GetPropInfo(AClass: TClass; const PropName: string;  
  AKinds: TTypeKinds = []): PPropInfo; overload;  
function GetPropInfo(TypeInfo: PTypeInfo;  
  const PropName: string): PPropInfo; overload;  
function GetPropInfo(TypeInfo: PTypeInfo; const PropName: string;  
  AKinds: TTypeKinds): PPropInfo; overload;
```

GetPropInfo 有四种重载形式。这四种函数都有三个参数:

参数 Instance、AClass、TypeInfo 被用来取得 RTTI。Instance 是一个类实例, AClass 是一个类引用类型实例(关于类引用类型可以参看第 9 章的“新颖的类工厂”一节), TypeInfo 可以调用函数 TypeInfo 来取得。例如:

```

if GetPropInfo(Button1, 'Tag') <> nil then
{ 或者 GetPropInfo(TButton, 'Tag') <> nil then }
{ 或者 GetPropInfo(TypeInfo(TButton), 'Tag') <> nil then }
{ 或者 GetPropInfo(TButton.ClassInfo, 'Tag') <> nil then }
{ 或者 GetPropInfo(Button1.ClassInfo, 'Tag') <> nil then }
ShowMessage('Tag 属性存在');

```

当使用类引用实例时，也可以这样：

```

{ 首先定义一个 TButton 的类引用类型 TButtonClass }
type TButtonClass = class of TButton;
{ 声明一个类引用类型变量 ButtonClass }
var
    ButtonClass: TButtonClass;
begin
{ 初始化变量 ButtonClass }
    ButtonClass := TButton;
    if GetPropInfo(ButtonClass, 'Tag') <> nil then
        ShowMessage('该属性存在');
end;

```

参数 PropName 是一个字符串，用于指定一个属性的名字。

参数 AKinds 用于指定属性类型。TTypeKinds 定义如下：

```

TTypeKinds = set of TTypeKind;
TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,
tkString, tkSet, tkClass, tkMethod, tkWChar, tkLString, kWString,
tkVariant, tkArray, tkRecord, tkInterface, tkInt64, tkDynArray);

```

当指定了 AKinds 参数，PropName 属性必须存在且其数据类型在 AKinds 指定的范围内时，GetPropInfo 返回值才会为非 nil。当 AKinds 为缺省值[]时，表示对属性的数据类型没有限制，可以是任何类型。

使用函数 IsPublishedProp：

```

function IsPublishedProp(Instance: TObject; const PropName: string):
    Boolean; overload;
function IsPublishedProp(AClass: TClass; const PropName: string):
    Boolean; overload;

```



你不要望文生义，以为还有 IsProp、IsPublicProp 等方法可以操作所有域的属性，其实 IsPublishedProp 是这样实现的：

```
function IsPublishedProp(Instance: TObject; const PropName: string):
    Boolean;
begin
    Result := GetPropInfo(Instance, PropName) <> nil;
end;
```

也就是说，上面 GetPropInfo 和 IsPublishedProp 只能判断一个 published 属性是否存在。

(2) 读取某个存在的属性的当前值：

GetOrdProp：用于原始类型属性，包括 integer、character、Boolean 和 subrange 等 Ordinal 类型。

GetEnumProp：用于枚举类型。

GetSetProp：用于集合类型。

GetObjectProp：用于对象类型。

GetStrProp：用于字符串类型。

GetWideStrProp：用于宽字符串类型。

GetFloatProp：用于浮点类型。

GetVariantProp：用于可变类型。

GetMethodProp：用于方法类型。

GetInt64Prop：用于 Int64 类型。

GetInterfaceProp：用于接口类型。

GetPropValue：用于任何类型，返回 Variant。

如果要设置某个属性的值，将上面所列函数和过程的“Get”变成“Set”后，就得到设置属性值的对应函数和过程。

一个读取和设置属性值的函数或者过程都具有类似如下的形式：

```
function GetOrdProp(Instance: TObject; const PropName: string):
    Longint; overload;
function GetOrdProp(Instance: TObject; PropInfo: PPropInfo):Longint;
    overload;
procedure SetOrdProp(Instance: TObject; const PropName: string;
    Value: Longint); overload;
procedure SetOrdProp(Instance: TObject; PropInfo: PPropInfo;
    Value: Longint); overload;
```

其中参数 PropInfo 就是函数 GetPropInfo 的返回值。

要特别注意的是，以上属性存取函数和过程在执行前不会首先判断该属性是否存在。如果属性不存在，将会产生异常。所以在调用它们前，首先应该使用函数 GetPropInfo 或者 IsPublishedProp 来判断属性是否存在，只有属性存在时才调用属性存取函数和过程。

(3) 其他衍生方法：

```
function GetEnumeratorName(TypeInfo: PTypeInfo; Value: Integer): string;
```

得到枚举值对应的字符串。注意这个函数只能用于非自定义顺序的枚举类型，因为自定义顺序的枚举类型没有 RTTI。

```
function GetEnumeratorValue(TypeInfo: PTypeInfo; const Name: string):  
Integer;
```

通过枚举值对应的字符串找到枚举值的顺序。

```
function GetObjectPropClass(Instance: TObject;  
    const PropName: string): TClass;  
function GetObjectPropClass(Instance: TObject; PropInfo: PPropInfo):  
    TClass; overload;  
function GetObjectPropClass(PropInfo: PPropInfo): TClass; overload;
```

得到对象属性的类型，返回一个类引用。

```
function SetToString(PropInfo: PPropInfo; Value: Integer;  
    Brackets: Boolean = False): string;
```

将集合用字符串表示，各项用逗号分割。Brackets 表示返回的字符串是否用方括号包裹。

```
function GetPropList(AObject: TObject; out PropList: PPropList):  
    Integer; overload;  
function GetPropList(TypeInfo: PTypeInfo; out PropList: PPropList):  
    Integer; overload;  
function GetPropList(TypeInfo: PTypeInfo; TypeKinds: TTypeKinds;  
    PropList: PPropList; SortList: Boolean = True): Integer; overload;
```

得到一个对象或者类的所有属性的列表。返回的列表在输出参数 PropList 中。函数返回值是列表的项数，即属性个数。

最后，我们举一些例子来展示使用 RTTI 所能获得的强大功能：

(1) 如何给父空间 Panel1 上的所有数据敏感控件指定 DataSource 属性？

```
procedure SetAllDataSource(DataSource: TDataSource; Parent: TWinControl);
```



```

var
  I: Integer;
  tFoundContrl: TControl;
  PropInfo: PPropInfo;
const
  PropName = 'DataSource';
begin
  for I := 0 to Parent.ControlCount-1 do
  begin
    tFoundContrl := Parent.Controls[I];
    PropInfo := GetPropInfo(tFoundContrl, PropName);
    {或者 PropInfo := GetPropInfo(tFoundContrl.ClassInfo, PropName);}
    if (PropInfo <> nil) and
      (GetObjectPropClass(PropInfo) = TDataSource) then{*}
      SetObjectProp(tFoundContrl, PropName, DataSource);
    end;
  end;
end;

```

对于*这句,这里是判断该属性类型是否为 TDataSource 类型本身,如果你自己扩展了 TDataSource 类(即从 TDataSource 直接或者间接继承),要求该属性类型是 TDataSource 本身或者其子类,那么可以改为:

```
GetObjectPropClass(PropInfo).InheritsFrom(TDataSource)。
```

InheritsFrom 是 TObject 的一个类方法,具体参看下面第(4)例。

(2) 如何得到对象或者类的所有属性名?

```

var
  I, J: Integer;
  PropList: PPropList;
begin
  J := GetPropList(ListBox1, PropList);
  {取得 ListBox1 所有属性放到 PropList 中}
  for I := Low(PropList^) to J-1 do
  begin
    ListBox1.Items.Add('第' + IntToStr(PropList[I]^NameIndex) +
      '个:' + PropList[I]^Name);
    end;
  end;
end;

```

其中 PPropList 是这样定义的：

```
PPropList = ^TPropList;
TPropList = array[0..16379] of PPropInfo;
PPropInfo = ^TPropInfo;
TPropInfo = packed record
  PropType: PTypeInfo; { 属性类型}
  GetProc: Pointer;    { 属性的 Get 方法地址}
  SetProc: Pointer;    { 属性的 Set 方法地址}
  StoredProc: Pointer; { 属性 Stored 方法地址}
  Index: Integer;      { 属性存取方法索引}
  Default: Longint;    { 属性默认值}
  NameIndex: SmallInt; { 属性名在属性列表中的索引}
  Name: ShortString;   { 属性名}
end;
```

首先得讨论一下 GetProc、SetProc 和 StoredProc 这三个字段。

我们可以用 Integer(GetProc)得到指针 GetProc 对应的地址，返回 4 个字节，进一步可用函数 IntToHex 将这个地址转化为十六进制字符串查看。如果地址：

最高字节(第三字节)为 FF，则表示该属性不存在对应的 Get/Set/Stored 方法，即 read/write/stored 都是对字段而不是使用方法存取属性。这时候，低三位是该字段对于对象实例(Instance)指针地址(即 Integer(Button1)、Integer(Label1))的偏移量。

因此，表达式：

```
Integer(GetProc)+ Integer(Instance)
```

的运算结果就是该属性对应字段的实际地址。我们可以通过这个地址取得属性的当前值。具体方法是：假设该属性是 TPropClass 类型，那么表达式：

```
TPropClass(Integer(GetProc)+ Integer(Instance))
```

或者：

```
TPropClass(Integer(SetProc)+ Integer(Instance))
{ 因为此时存取都是通过字段进行，因此，GetProc 和 SetProc 实际都是指向同一个地址：字段的存储位置 }
```

可以得到属性的值。

最高字节(第三字节)为 FE，则表示 GetProc 是一个 virtual 方法。低三位表示该方法对于实例地址的偏移。也就是说，此时实例地址是虚拟方法表(VMT)的入口地址，低三位即方法在 VMT 中的索引。Integer(GetProc)+ Integer(Instance)即是 GetProc 方法的实际地址。

最高字节(第三字节)小于 FE，则表示 4 个字节共同构成方法 GetProc 的地址。

对于 SetProc 和 StoredProc，地址规则和上面的 GetProc 是相同的。

(3) 如何得到枚举对应的字符串，或者得到字符串对应的枚举值？

```

type
  TEnum = (emString1, emString2, emString3);
Implementation
uses TypInfo;
procedure TForm1.Button1Click(Sender: TObject);
var
  Enum: TEnum;
  S: String;
  I: Integer;
begin
  Enum := emString2;
  S := GetEnumName(TypeInfo(TEnum), Ord(Enum));
  I := GetEnumValue(TypeInfo(TEnum), S);    {I := Ord(Enum);}
  ShowMessage(S);
  ShowMessage(IntToStr(I));
end;

```

(4) 关于 is 和 as。它们也是依赖于 RTTI 工作的。

它们的使用定义如下：

```
object is/as class
```

假如对象实例 object 的类型是 class 本身或者其子类，返回 True，否则(或者 object=nil)返回 False。
as 将 object 转化为 class 类型后返回。如果 object is class=False，则 as 转化异常。所以 as 以前一般都要先 is，除非你能肯定 is 返回 True。

is 和 InheritsFrom 的比较：

TObject 提供一个类方法：

```
class function InheritsFrom(AClass: TClass): Boolean;
```

和 is 的含义是一样的，is 内部实现就是调用这个方法。但是 InheritsFrom 还可以由类调用，因为它是类方法，而 is 不能。

as 和 Typecasts (强制类型转换)的对比：

Typecasts 可以转化任何类型的值或者变量到另一个类型，转化规则依赖于两个类型的关系。一些复杂的转化能否成功不是确定的，即使编译通过了，运行时也可能异常。对于对象的转化，一般也配合 is 使用。比如：

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

或者：

```
if ActiveControl is TEdit then (ActiveControl as TEdit).SelectAll;
```

推荐使用 Typecasts 方法，是因为前面 is 已经确定类型匹配。as 使用 RTTI，比 Typecasts 慢得多。

(5) 我们从 TObject.ClassInfo 知道 TObject 已经具有处理 RTTI 的能力，但是 TObject 本身并不生成 RTTI，因为它没有打开生成 RTTI 的编译指令。TObject 的直接子类 TPersistent 才真正具有 RTTI 的能力，那么根据继承的原则，TPersistent 的直接子类和间接子类都具有 RTTI。

如果我从 TObject 直接继承一个子类，并要求具有 RTTI 能力，应该怎么办？

在类声明前后加上编译指令 \$M：

```
type

{$M+}

TMyObject = class(TObject)
private
    FExProp: String;
published
    property ExProp: String read FExProp write FExProp;
end;

{$M-}

implementation

uses TypInfo;

procedure TForm1.Button1Click(Sender: TObject);
var
    MyObject: TMyObject;
begin
    MyObject := TMyObject.Create;
    if GetPropInfo(MyObject, 'ExProp') <> nil then
        ShowMessage(GetStrProp(MyObject, 'ExProp'));
    FreeAndNil(MyObject);
end;
```



小结

本小节介绍了运行时类型信息的获取方法，并详细讲解了如何运用运行时类型信息来存取对象的属性值。这些方法是我们提升程序功能和灵活性的重要手段。

5.2 VCL 的消息机制

Windows 是一个基于消息传递和处理的操作系统，如果 VCL 组件不能处理消息，那么就只能是一堆无用的代码，好比一大桌子塑料做的菜，不能食用。

VCL 消息处理是一个比较复杂的过程，但是在本节里，你会发现搞明白它不过是一件轻而易举的事情。VCL 消息处理过程虽然是只张牙舞爪的大蟹，但是我将它作成了一道好菜放在盘子里给你端上来，你还会怕面对它吗？

5.2.1 VCL 消息机制

Windows 是一个基于消息的操作系统。当你的应用程序运行时，程序中每个有窗口的控件（TWinControl，也就是有句柄的控件）都会在 Windows 中注册一个窗口过程，这个过程在 VCL 中叫作 MainWndProc。Windows 消息到达 MainWndProc 后，由 MainWndProc 调用一系列方法处理这个消息。

VCL 消息机制的整个流程如下所示：

```
Windows->Delphi Application-> TWinControl
MainWndProc->WndProc->Dispatch->Handler
```

下面详细分析这个流程。

首先要搞清楚的是，Windows 消息是如何到达 VCL 的 MainWndProc 的。每个 Delphi Application 是一个 TApplication（一个特殊的 VCL 类）的实例，它运行（Application.Run）后调用方法 ProcessMessages，实现一个消息循环。

(1) ProcessMessages 内部循环调用另一个私有方法 ProcessMessage。从而可以不断地从 Windows 消息队列中提取属于本 Application 的消息（如果是 WM_QUIT，则应用程序终止，不进行下面的流程），每提取到一个消息，就触发事件 OnMessage 并调用方法 DispatchMessage。ProcessMessage 的实现代码如下（已经简化）：

```
function TApplication.ProcessMessage(var Msg: TMsg): Boolean;
var
    Handled: Boolean;
begin
    Result := False;
    if PeekMessage(Msg, 0, 0, 0, PM_REMOVE) then
```

```

begin
  Result := True;
  if Msg.Message <> WM_QUIT then
  begin
    Handled := False;
    { 如果事件 OnMessage 有代码存在，则执行这段代码 }
    if Assigned(FOnMessage) then FOnMessage(Msg, Handled);
    if not Handled then
    { 如果 OnMessage 没有处理该消息，则派遣消息到参数 Msg 指定的窗口过程，以继续消息传递流程 }
      DispatchMessage(Msg);
    end
  else
    { 如果 Msg.Message = WM_QUIT，则终止程序运行 }
    FTerminate := True;
  end;
end;

```

(2) 事件 OnMessage 中，如果有用户书写的代码，那么执行。

(3) DispatchMessage。它是一个 API 函数，其原型如下：

```
function DispatchMessage(const lpMsg: TMsg): Longint; stdcall;
```

它需要一个 TMsg 类型的参数 lpMsg。lpMsg 是一个记录格式的数据，包含了消息接收者的句柄和消息正文。这样，DispatchMessage 就根据参数 lpMsg 的接收者句柄字段，将消息正文发送给该句柄对应的窗口控件在 Windows 中注册的窗口过程：TWinControl.MainWndProc。所以到这里，消息就传递给 TWinControl 了。这个过程叫作消息派遣。不过这里是在 Windows 系统内派遣，还没有进入控件内部。当消息进入控件内部时，我们还会看到另一个派遣过程。

下面，消息进入 TWinControl 对象的内部循环实现一系列处理。参考本节头部的消息流程图，我们首先分析 TWinControl.MainWndProc。

1. 窗口过程 MainWndProc

实现如下：

```

procedure TWinControl.MainWndProc(var Message: TMessage);
begin
  try
    try
      { 从下面一行代码可以看到，消息被直接传入方法类型属性 WindowProc。MainWndProc

```

```

    本身未对消息作任何特别处理，它不过是充当一个中转站的角色}
    WindowProc(Message);
  finally
    FreeDeviceContexts;
    FreeMemoryContexts;
  end;
except
  Application.HandleException(Self);
end;
end;

```

在 TWinControl 创建实例时，自动注册这个窗口过程 MainWndProc。窗口过程的含义是：注册者将消息交给这个过程处理。如果你开发不从 TWinControl 或其子类派生组件，并希望这个组件能够处理消息，那么必须自定义窗口过程并注册，详情见第 9 章的“自定义窗口过程”一节。

MainWndProc 调用了 WindowProc。WindowProc 是从 TControl 继承的一个方法指针类型的属性，TControl 中定义：

```

TControl = class(TComponent)
private
  FWindowProc: TWndMethod;
  {Classes 单元定义:TWndMethod = procedure(var Message: TMessage) of object;}
  .....
protected
  procedure WndProc(var Message: TMessage); virtual;
  {可见 WndProc 和 FWindowProc 是兼容的}
  .....
public
  {WindowProc 被声明为一个运行时属性}
  property WindowProc: TWndMethod read FWindowProc write FWindowProc;
  .....
end;

```

一个 TControl 实例被初始化时指定：

```

constructor TControl.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FWindowProc := WndProc;
  .....
end;

```

所以 TWinControl.MainWndProc 中的：

```
WindowProc(Message)
```

等价于调用 WndProc(Message)。

分析到这里，我们发现窗口过程 MainWndProc 和属性 WindowProc 不过都是将 API 函数 DispatchMessage 转发的消息 Message 倒来倒去，最终送到虚拟方法 WndProc 口中，它们本身都没对消息做任何处理。

2. 虚拟方法 WndProc

WndProc 是 TControl 在 protected 区定义的虚方法，有自己的处理代码，其子类 TWinControl 又进行了覆盖。

WndProc 让控件能对一些特殊的消息进行必要响应。如：焦点 (focus)、鼠标 (mouse)、键盘 (keyboard) 等类消息。最终，所有无意义的消息被作必要处理后抛弃，其余消息被传入方法 Dispatch，Dispatch 实现消息派遣。

3. 虚拟方法 Dispatch

Dispatch 是从 TObject 继承下来的虚拟方法：

```
TObject = class
  procedure Dispatch(var Message); virtual;
  .....
end;
```

Dispatch 首先在本类中查找相应的消息方法（也就是上面消息流程图中的 Handler），如果没找到，那么逐级上溯父类、祖父类，直到找到对应的消息方法。如果搜索完所有类（即 TObject 的搜索也完成了）还是不能找到满足条件的 Handler，那么调用缺省消息处理方法 DefaultHandler，否则将消息传送给找到的消息方法，这个过程也叫作消息派遣。和前面讲过的 DispatchMessage 不同，这次是在 Application 中实现派遣过程。

```
procedure TObject.Dispatch(var Message);
begin
  { 搜索 Handler }
  .....
  { 如果没找到则跳转到 default }
  JE      @@default
  @@default:
    POP    ESI
    MOV    ECX, [EAX]
```

```

    JMP     dword ptr [ECX].vmtDefaultHandler
    {vmtDefaultHandler 即 VMT 负偏移-16 处, 该处存放了方法入口:
    DefaultHandler(var Message)}
end;

```

4. 虚拟方法 DefaultHandler

DefaultHandler 是从 TObject 继承下来的虚方法, 目的是保证所有消息都能得到处理。一些 TObject 的子类对它作了覆盖, 如: TControl、TWinControl, 从而实现不同类对消息需要作的不同缺省处理。

至此, VCL 消息机制的整个流程我们已经分析完毕。

小结

本小节详细分析了 VCL 的消息机制, 对整个消息传递和处理流程作了全面解析。

VCL 消息处理机制是让 VCL 动起来的基础, 不能处理消息的类和对象是“死”的, 是根本不能应用的, 更不要说用它们来开发多姿多彩、功能强大的程序和软件。如果说 VCL 架构是 VCL 体系的骨, 那么 VCL 消息机制则是这个骨架上的血肉。

值得说明的是, 有窗口句柄并不是一个控件能处理消息的必要条件。

我们观察本节开头的流程图, 如果一个消息不是 Application 从消息队列取得, 换句话说, 如果消息不需要 Application.DispatchMessage 来派遣, 而是直接进入 TWinControl.MainWndProc 或者更后面的阶段, 那么窗口句柄显然是不必要的。只有 Application.DispatchMessage 派遣消息时才需要一个窗口句柄, 该句柄描述了消息接收者。

这样, 在 TControl 内部, 本身就可以实现一个消息传递和处理流程。

5.2.2 处理消息的八种方法

从 VCL 消息处理流程可知, 我们可以在一些事件中或者通过覆盖虚方法实现消息处理。当然还可以编写 Windows Hook 程序来处理消息, 不过这些方法不在本书讨论之列。

下面开始讨论如何在 Delphi 程序中截获和处理控件消息。归纳起来, 大致有如下一些方法:

1. 使用事件 Application.OnMessage

在这个事件里可以处理属于本应用程序的、除 WM_QUIT 以外的所有消息。不过千万要注意“所有”二字! 在实际应用中, 我们往往只是处理一个或者几个消息, 如果在这个事件中实现个别消息的处理, 则很明显效率很低。因为你写的代码对每个消息都要作出判断, 而 Windows 消息又是奇多无比, 这就好像我们在大海里捞针, 非常划不来。

不过我们还是写几行代码来演示这种方法:

```

type
    TForm1 = class(TForm)

```

```

    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    procedure HandleAppMsg(var Msg: TMsg; var Handled: Boolean);
  end;

  procedure TForm1.HandleAppMsg(var Msg: TMsg; var Handled: Boolean);
  begin
    if Msg.message = WM_KEYDOWN then
    begin
      ShowMessage('OnMessage');
      { 在 OnMessage 中处理一个消息后应该将参数 Handled 置为 True，否则
        DispatchMessage 会派遣该消息，将已经处理了的消息继续传递下去 }
      Handled := True;
    end;
  end;

  procedure TForm1.Button1Click(Sender: TObject);
  begin
    { 发送一个消息到消息队列。注意这里不要使用 SendMessage。因为 SendMessage 发送的
      消息直接进入窗口而没有进入消息队列。Application 是不知道没有进入消息队列的消息
      的，所以 OnMessage 也就根本不能截获这个消息。 }
    { 因为只是为了说明原理，所以 WM_KEYDOWN 的两个参数都简单地使用了 0，在实际应用中，
      这样一个消息当然是毫无用处的 }
    PostMessage(Handle, WM_KEYDOWN, 0, 0);
  end;

  procedure TForm1.FormCreate(Sender: TObject);
  begin
    { 用 HandleAppMsg "挂住" : Application.OnMessage }
    Application.OnMessage := HandleAppMsg;
  end;

```

通过 Application 截获消息只有一个方法：使用 Application.OnMessage。

以下开始讲如何在控件窗口内部截获消息。

在运行下面的例子时注意要将：

```
Application.OnMessage := HandleAppMsg;
```

或者：

```
Handled := True;
```

注释掉，否则消息不能传入控件窗口。那样你就会怀疑我是不是在胡说八道了！

2. 覆盖虚拟方法 WndProc

```
type
  TForm1 = class(TForm)
  protected
    procedure WndProc(var Message: TMessage); override;
  end;

procedure TForm1.WndProc(var Message: TMessage);
begin
  if Message.Msg = WM_KEYDOWN then
    ShowMessage('WndProc')
  else inherited;
  { 如果是别的消息，则应该调用父类的处理方法继续处理，从而用 Dispatch 实现消息派遣 }
end;
```

运行上面和接下来的几个例子时，可以用两个方法来生成 WM_KEYDOWN 消息：

- (1) 使用 PostMessage 显式发送 WM_KEYDOWN。
- (2) 当焦点在窗体上时（注意不是焦点在窗体某控件上），按下键盘某键。

3. 嫁接属性 WindowProc

这种方法和覆盖 WndProc 本质是一样的。因为上节我们讲了：方法 WndProc 和属性 WindowProc 其实是一个东西。

```
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    OldWindowProc: TWndMethod;
  protected
    procedure NewWindowProc(var Message: TMessage);
  end;

procedure TForm1.NewWindowProc(var Message: TMessage);
begin
```

```

    if Message.Msg = WM_KEYDOWN then
        ShowMessage('WindowProc')
    else if Assigned(OldWindowProc) then
        { 下面的代码意思相当于"覆盖虚拟方法WndProc"中的"else inherited;" }
        OldWindowProc(Message);
    end;

    procedure TForm1.FormCreate(Sender: TObject);
    begin
        { 用OldWindowProc将WindowProc保存 }
        OldWindowProc := WindowProc;
        { 给WindowProc指定新的值NewWindowProc }
        WindowProc := NewWindowProc;
    end;

```

4. 覆盖虚拟方法 Dispatch

```

type
    TForm1 = class(TForm)
    protected
        procedure Dispatch(var Message); override;
    end;

    procedure TForm1.Dispatch(var Message);
    begin
        if TMessage(Message).Msg = WM_KEYDOWN then
            ShowMessage('Dispatch')
        else inherited;      { 其他的消息调用父类的Dispatch 实现派遣 }
    end;

```

5. 覆盖虚拟方法 DefaultHandler

```

type
    TForm1 = class(TForm)
    protected
        procedure DefaultHandler(var Message); override;
    end;

```




```

procedure TForm1.DefaultHandler(var Message);
begin
    if TMessage(Message).Msg = WM_KEYDOWN then
        ShowMessage('DefaultHandler')
    else inherited;
end;

```

覆盖 DefaultHandler 来截获消息在理论上是可行的，但是非常非常不保险。比如上面这段代码就没有起作用，根本没有显示“DefaultHandler”，也就是说，Dispatch 根本没有将消息传给 DefaultHandler。我们知道 Dispatch 是要找消息方法——在本类和所有的父类中找，这就麻烦了，我如果要用 DefaultHandler 处理一个特定消息，就需要保证本类和所有父类都没有对应的消息方法——这简直是开玩笑！难道让我去将 VCL 源代码中的消息方法统统删除？因为在 TForm1 的父类中已经有 WM_KEYDOWN 的消息方法，所以 TForm1.DefaultHandler 根本接收不到这个消息。

所以，DefaultHandler 对于处理特定消息并没有应用价值；实际上，它的目的是给所有被别人抛弃的孤儿们一口饭吃，而不是要照顾某个特别的孩子。

以上五种方法都是自己写代码通过对各种消息作类型判断来截获和处理特定消息的。客观地说，效率都是不高的，尤其是 Application.OnMessage。因为它们要求每个到来的消息都要停下来接收它们的检查，就好像在高速公路上设置路卡一样，显然很讨人厌。

尽管以上五种方法有降低效率的缺点，但是使用它们也可以在一段代码中处理多个、大量消息。

下面我们再讲三种处理方法。和已经讲过的方法不同的是，这三种方法都是只对特定类型的消息作检查。因为这三种方法的处理点都在 VCL 消息处理流程中的 Dispatch 后，这时候，Dispatch 已经对每个消息作了派遣，消息们各奔东西，我们不再是在火车站、机场围堵大量到来的消息，而是专门到特定消息下榻的宾馆去拜访它们。

因为下面的三种方法是对特定消息的专程拜访，所以当在一个类中需要处理大量消息时，就需要定义比较多的方法、在很多地方写大量代码。

在运行以下三种方法的例子时，请首先去掉已讲几种方法的代码，因为它们接收到 WM_KEYDOWN 消息后就将它暗杀了，接下来的消息流程不能和这个消息见面。

6. 添加消息方法

```

type
    TForm1 = class(TForm)
    protected
        procedure WMKeydown(var Message: TMessage); message WM_KEYDOWN;
    end;

procedure TForm1.WMKeydown(var Message: TMessage);
begin

```

```
ShowMessage('message WM_KEYDOWN');
inherited;    { 调用父类的消息方法处理消息 }
end;
```

7. 覆盖事件驱动方法（如果有）

```
type
  TForm1 = class(TForm)
    protected
      procedure KeyDown(var Key: Word; Shift: TShiftState);override;
    end;

  procedure TForm1.KeyDown(var Key: Word; Shift: TShiftState);
  begin
    ShowMessage('KeyDown');
    inherited;    { 调用父类的事件驱动方法处理消息 }
  end;
```

8. 使用事件

对于设计时可见的控件，使用事件来处理消息是再简单不过的，就是在 Object Inspector 中定位到相应事件，然后写些代码就可以。

如果是在运行时使用事件，则要麻烦一点。要使用一种叫作“嫁接”的办法。关于嫁接我实在不想浪费纸张再讲了，因为上面在讨论利用属性 WindowProc 处理消息时已经非常充分地展现了嫁接术的全过程。但是为了保持内容的完整性，我还是写几行代码吧：

```
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    OldOnKeyDown: TKeyEvent;
    procedure NewOnKeyDown(
      Sender: TObject; var Key: Word; Shift: TShiftState);
    end;

  procedure TForm1.NewOnKeyDown(
    Sender: TObject; var Key: Word; Shift: TShiftState);
  begin
```

```

ShowMessage('OnKeyDown');
if Assigned(OldOnKeyDown) then
  OldOnKeyDown(Sender, key, Shift);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  OldOnKeyDown := OnKeyDown;
  OnKeyDown := NewOnKeyDown;
end;

```

好了，八种消息处理方法讲完了，不知道大家有多少收获。

小结

本小节详细讲述了 VCL 中处理消息的八种方法。详细分析了八种消息处理方法的优缺点。尽管后三种方法只能对特定消息进行处理，在处理多个消息时有多处撰写代码的缺点，但是它们的处理效率毕竟高多了。所以我们强烈推荐使用后三种方法处理消息，反对使用前五种。

本小节的重点是：消息方法、事件驱动方法、事件嫁接。

事件嫁接是一种很值得重视的编程技巧。嫁接的原理对于所有可见方法也是适用的。如果希望在一个类中处理另一个类的消息，嫁接是一种极好选择。

5.2.3 选用什么方法发送消息

在 VCL 中，有很多种发送消息的方法，另外，也有一些 API 函数可以用来发送消息。

下面我们分别讲述这两类发送消息的方法。

1. VCL 的消息发送方法

VCL 的消息发送方法定义在类中，也就是说，它们是由类实例调用的，而不同于 API 函数。这些方法的消息接收者也是固定的（一般是自己），不能指定。

（1）调用保护方法：

```

procedure TWinControl.MainWndProc(var Message: TMessage);

```

MainWndProc 是 TWinControl 的一个保护方法，所以这个方法的调用者必须是本类的对象（即 Self）。如：

```

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);

```

```
protected
    procedure WMKeyDown(var Message: TMessage); message WM_KEYDOWN;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    Msg: TMessage;
begin
    { 初始化记录Msg，全部清零 }
    FillChar(Msg, SizeOf(Msg), #0);
    Msg.Msg := WM_KEYDOWN;
    MainWndProc(Msg);
end;

procedure TForm1.WMKeyDown(var Message: TMessage);
begin
    ShowMessage('message WM_KEYDOWN');
    inherited;
end;
```

(2) 使用属性：

```
TControl.WindowProc: TWndMethod;
```

我们不具体举例子了，将上面的：

```
MainWndProc (Msg) ;
```

改为：

```
WindowProc (Msg) ;
```

即可。

(3) 调用虚拟方法：

```
procedure TControl.WndProc(var Message: TMessage); virtual;
```

和上面的方法是类似的，将：

```
WindowProc (Msg) ;
```

改为：

```
WndProc (Msg);
```

即可。

(4) 调用公开方法：

```
function TControl.Perform(
    Msg: Cardinal; WParam, LParam: Longint): Longint;
```

这个方法内部是如下实现的：

```
function TControl.Perform(
  Msg: Cardinal; WParam, LParam: Longint): Longint;
var
  Message: TMessage;
begin
  Message.Msg := Msg;
  Message.WParam := WParam;
  Message.LParam := LParam;
  Message.Result := 0;
  if Self <> nil then WindowProc(Message);
  { 注意上一句, 消息不进入 MainWndProc, 当然 Application.OnMessage 更是蒙在鼓里 }
  Result := Message.Result;
end;
```

以上 4 种方法都是给 TControl 对象本身发送消息，所以没有用于指定消息接收者的 Handle 参数。

(5) 调用公开方法：

```
procedure TWinControl.Broadcast(var Message);
```

顾名思义，Broadcast 用来广播消息，它的接收者是调用者的所有子控件。我们可以看这个方法的内部实现：

```
procedure TWinControl.Broadcast(var Message);
var
  I: Integer;
begin
  for I := 0 to ControlCount - 1 do
  begin
    Controls[I].WindowProc(TMessage(Message));
    { 消息也是直接进入 WindowProc }
    if TMessage(Message).Result <> 0 then Exit;
  end;
end;
```

2. API 函数发送消息

可以发送消息的 API 比较多，我们可以分为以下几类：

(1) 给指定窗口发送消息：

```
function SendMessage(hWnd: HWND; Msg: UINT;
  wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;
```

```
function DispatchMessage(const lpMsg: TMsg): Longint; stdcall;
function PostMessage(hWnd: HWND; Msg: UINT;
    wParam: WPARAM; lParam: LPARAM): BOOL; stdcall;
```

接收窗口由参数 hWnd 和 lpMsg.hwnd 指定。

SendMessage 和 DispatchMessage 功能是类似的,它们都是直接将消息发送给指定窗口,也就是说,它们发送的消息直接进入 TWinControl.MainWndProc。

但 PostMessage 是将消息放入消息队列。PostMessage 发送的消息会在整个消息流程中传递和处理。这个窗口可以属于当前应用程序,也可以属于其他应用程序。

(2) 给当前应用程序发送消息:

```
function SendAppMessage(Msg: Cardinal; wParam, lParam: Longint): Longint;
```

注意这个函数其实是 Delphi 自己定义的,并不是 API 函数,我们将它列到这里,只是为了说明方便。其内部是如此实现的:

```
function SendAppMessage(Msg: Cardinal; wParam, lParam: Longint): Longint;
begin
    if Application.Handle <> 0 then
        Result := SendMessage(Application.Handle, Msg, wParam, lParam)
    else
        Result := 0;
end;
```

可见 SendAppMessage 并没有多么神奇,它不过是内部调用 SendMessage 给 Application 发送消息。

TApplication 虽然是从 TComponent 继承而不是 TWinControl 的子类,但是它也定义了一个窗口(不过和 TWinControl 的窗口相比,Application 的窗口显示方式很不同)。你运行 Delphi,随便新建一个简单工程,然后按 F9 键,在 Windows 标题栏上,你会发现一个标题叫做“Project1.exe”的条,那就是 Application 的窗口。这个标题可以通过属性 Application.Title 改变。Application.Handle 就是这个窗口的句柄。

使用 SendAppMessage 或者直接使用 SendMessage 给 Application 发送的消息可以在 Application.OnMessage 中接收。

(3) 给线程发送消息:

```
function PostThreadMessage(idThread: DWORD; Msg: UINT;
    wParam: WPARAM; lParam: LPARAM): BOOL; stdcall;
```

其中 idThread 可用 API 函数获得:

```
function GetWindowThreadProcessId(
    hWnd: HWND; lpdwProcessId: Pointer = nil): DWORD; stdcall; overload;
```

```
function GetWindowThreadProcessId(  
    hWnd: HWND; var dwProcessId: DWORD): DWORD; stdcall; overload;
```

GetWindowThreadProcessId 取得创建窗口 hWnd 的线程和进程，线程代号作为函数返回值返回，进程代号由第二个参数传回。

如果 hWnd 指定的是某个应用程序的主窗口，那么返回值是该应用程序的主线程和主进程。从而可以在一个应用程序中给另一个应用程序发送消息。相关例子请参看第 9 章的“自定义系统惟一消息”一节。

小结

本小节全面介绍了在 Delphi 程序中发送各种消息的不同方法。发送消息的方法主要分为两类：

(1) VCL 类提供的方法。这类方法主要用于给控件自身和对象的子控件发送消息，而且接收对象可以是没有窗口的。

(2) API 函数发送消息。这类方法可以对任何窗口发送消息。

本节的重点是：

(1) TControl.Perform。

(2) SendMessage、PostMessage 以及二者的区别。

5.2.4 VCL 消息大全

VCL 消息包含两大类：Windows 标准消息和 VCL 自定义消息。Windows 标准消息是 Windows 自己定义的，供 Windows 标准控件使用(这些控件主要分布在 Delphi 组件面板的“Standard”和“Win32”页)；VCL 自定义消息供 VCL 自定义控件使用。

Windows 标准消息按控件类型主要划分为 23 类(在 Win32 SDK Reference 的键入索引“System-Defined Messages”可以看到)。Delphi 的 Messages 单元声明了它们。每类消息用不同的头部标示：

ABM	Application desktop toolbar	应用程序窗口(显示在 Windows 工具栏)使用
BM	Button control	TButton 使用
CB	Combo box control	TComboBox 使用
CDM	Common dialog box	TOpenDialog、TSaveDialog 等通用对话框使用
DBT	Device	计算机物理设备使用
DL	Drag list box	TListBox 处理拖动消息
DM	Default push button control	API 函数 MessageBox 等生成的对话框使用
EM	Edit control	TEdit 以及 TMemo、TRichEdit 等编辑框使用
HDM	Header control	THeaderControl 使用
LB	List box control	TListBox 使用

LVM	List view control	TListView 使用
PBM	Progress bar	TProgressBar 使用
PSM	Property sheet	属性页使用
SB	Status bar window	TStatusBar 使用
SBM	Scroll bar control	TScrollBar 使用
STM	Static control	TStaticText、TImage 等静态控件使用
TB	Toolbar	TToolBar 使用
TBM	Trackbar	TTrackBar 使用
TCM	Tab control	TTabControl 使用
TTM	Tooltip control	Hit 窗口使用
TVM	Tree-view control	TTreeView 使用
UDM	Up-down control	TUpDown 使用
WM	General window	通用窗口消息。所有窗口都可以使用，这类消息是最常用的

还有一类消息，叫作通知消息（Notification Messages）或者通知消息码（Notification Messages Code/ID）。一般地，不同种类的控件有相应的通知消息，常常是在控件类型标记后加上字母“N”标识。如：BN 对应按钮，LBN 对应列表框等。

通知消息主要用在父子控件环境中，用以保持父子的互动。如用户在按钮上单击鼠标时，按钮会向父控件（如 Panel、Form）发送 BN_CLICKED 消息。通知消息实际上是通过 WM_COMMAND 消息发给父控件的（不过滚动条例外），在该消息的 wParam 中含有通知消息码（如 BN_CLICKED）和控件的 ID，在 lParam 中则包含了控件的句柄。

除了上面的 Windows 标准消息外，VCL 为了满足自身需要，也自定义了许多 VCL 控件通用消息（一般用 CM_ 标识）、特定控件专用消息（以控件类别标识）以及通知消息，声明在对应控件所在单元。

下面对主要 Windows 标准消息和 VCL 自定义消息作个简单介绍。

1. 通用窗口消息

它们以 WM_ 开头。所有有窗口的控件(TWinControl)可以处理此类消息。它们在 Windows 消息大家庭中居于核心地位。在消息方法中，可用默认的 TMessage 类型接收这类消息；但是 VCL 也自定义了一些特定的消息类型(T+去掉下划线的消息名)，简化了消息参数的分解，所以也可以用它们来接收消息。

WM_CREATE

说明：窗口创建消息。窗口是通过 CreateWindowEx 或者 CreateWindow 创建的，在这两个函数创建了窗口后、返回之前发送这个消息。一般在创建非 VCL 窗体(不使用 TForm)和编写 DLL 中才会用到。



格式：0、Integer(PCreateStruct)。分别表示参数 WParam 和 LParam。0 表示该参数未使用或者保留为将来使用。下同。

WM_DESTROY

说明：窗口销毁消息，和 WM_CREATE 对应。在“剪贴板监视器”一节中我们用到了这个消息。使用 VCL 的 TForm 时，我们可以在事件 OnCreate 和 OnDestroy 中书写代码。

格式：0、0。

WM_MOVE

说明：当控件被移动后会接收到这个消息。

格式：0、MakeLong(xPos, yPos)。

WM_SIZE

说明：当控件大小改变后接收到这个消息。TControl.OnCanResize(大小改变前)和 TControl.OnResize(大小改变后)对应这个消息。TControl 有众多的保护区属性和方法，如 OnCanResize 和 OnResize，如果在你的控件中发布了它们，就可以直接使用，否则需要使用消息方法。

格式：fwSizeType、MakeLong(nWidth, nHeight)。

WM_ACTIVATE

说明：当窗体(TForm)在激活和非激活状态变化前发送和接收这个消息。对应 TForm. OnActivate 和 TForm. OnDeactivate。

格式：MakeLong(fActive, 1/0)、HWND。1/0 表示是否最小化。

WM_ACTIVATEAPP

说明：属于不同应用程序的两个窗体在激活和非激活状态变化。这两个窗体同时收到这个消息。

格式：1/0、另一个窗体所属线程的标示。

WM_SETFOCUS

说明：使控件获得输入焦点。和 API 函数 SetFocus 以及 TWinControl.SetFocus 功能相似。相反的是 WM_KILLFOCUS。

格式：HWND、0。

WM_ENABLE

说明：控件的 Enabled 属性发生变化。和 API 函数 EnableWindow (Handle)以及 TControl.Enabled 功能相似。

格式：1/0、0。

WM_FONTCHANGE

说明：控件 Font 属性发生变化。

格式：0、0。

WM_SETREDRAW

说明：设置控件是否需要重画。和 API 函数 InvalidateRect 以及 TControl.Invalidate 等功能相似。

格式：1/0、0。

WM_SETTEXT

说明：设置控件的 Text(Edit 类)或者 Caption(非 Edit 类)。

格式：0、Integer(PString)。

WM_GETTEXT

说明：取得控件的 Text(Edit 类)或者 Caption(非 Edit 类)。

格式：缓冲区长度、Integer(PString)。注意：发送该消息前你应该首先使用 GetMem 给 PChar 分配内存。

WM_GETTEXTLENGTH

说明：取得控件的 Text(Edit 类)或者 Caption(非 Edit 类)的长度。

格式：0、0。

WM_PAINT

说明：控件绘制。主要在非图形控件中使用。在图形控件中，一般覆盖虚拟方法 Paint(WM_PAINT 的消息方法内部调用它)。

格式：HDC、0。

WM_CLOSE

说明：窗口关闭或者应用程序结束。TForm.OnClose 对应它。

格式：0、0。

WM_QUERYENDSESSION

说明：当 Windows 需要关闭或者注销时，Windows 发给当前运行的程序(在主窗口接收)，询问是否可以关闭或者注销。如果程序返回 0，则不能关闭或者注销。然后系统再给该程序发送消息

WM_ENDSESSION, 告知是否会关闭/注销。

格式: 0、ENDSESSION_LOGOFF。

WM_ENDSESSION:

说明: 告知程序系统是否要注销/关闭。

格式: 1/0、ENDSESSION_LOGOFF。如果发送:

SendMessage(Application.Handle, WM_ENDSESSION, 1, ENDSESSION_LOGOFF);

可以关闭程序。和:

SendMessage(Application(或者 Application.MainForm).Handle, WM_CLOSE, 0, 0);

功能相似。

WM_QUIT

说明: API 函数 PostQuitMessage 内部发送这个消息, 告知应用程序退出消息循环并结束。

格式: nExitCode、0。

WM_QUERYOPEN

说明: 当窗体需要从图标状态(最小化)恢复到原来大小和位置。

格式: 0、0。

WM_ERASEBKGD

说明: 窗口重画的过程大体上是这样的: (1) 擦除需要重画的区域, 这个区域变为无效区域; (2) 重画无效区域。这个消息就是告知窗口擦除背景区域。

格式: HDC、0。

WM_SHOWWINDOW

说明: 显示/隐藏窗口。API 函数 ShowWindow 相似功能。

格式: 1/0、fnStatus。

WM_GETMINMAXINFO

说明: 窗口的大小或者位置将要改变。

格式: 0、Integer(PMinMaxInfo)。PMinMaxInfo 是一个记录, 通过它可以设置窗口最大、最小化时的大小、位置。

WM_NEXTDLGCTL

说明: 指定窗口上哪个控件接收焦点。

格式：wCtlFocus、1/0。当 lParam=1 时，wCtlFocus 应该是要接收焦点的控件句柄。当 lParam=0 时，如果 wCtlFocus=0，那么当前控件的下一个控件接收焦点，否则，上一个接收。

WM_DRAWITEM

说明：Windows 标准控件(如 ListBox、ComBox)绘制某部分。一般在对应事件 OnDrawItem 中处理。

格式：发送此消息的控件的标示、Integer(PDrawItemStruct)。

WM_MEASUREITEM

说明：Windows 标准控件(如 ListBox、ComBox)某部分的大小需要改变。一般在对应事件 OnMeasureItem 中处理。

格式：发送此消息的控件的标示、Integer(PMeasureItemStruct)。

WM_SETFONT

说明：更改控件字体。

格式：HFont、1/0。HFont 即 TFont.Handle，1/0 表示是否按照新的字体重新绘制。

WM_GETFONT

说明：取得控件字体。返回 HFont 即 TFont.Handle。

格式：0、0。

WM_SETHOTKEY

说明：设置应用程序级热键。

格式：MakeWord(vkey, modifiers)、0。MakeWord 构造一个 Word，vkey 表示虚拟键，modifiers 是修饰键(如 HOTKEYF_ALT、HOTKEYF_CONTROL、HOTKEYF_SHIFT、HOTKEYF_EXT)。

WM_HOTKEY

说明：设置系统级热键。该热键需要用 API 函数 RegisterHotKey 注册。当热键按下后，此消息被送到注册该热键的线程的消息队列顶部。

格式：热键标示、MakeWord(fuModifiers,uVirtKey)。

WM_GETHOTKEY

说明：取得热键。

格式：0、0。返回一个 Word，地位是 vkey，高位是 modifiers，可以用 Lo 和 Hi 分解。

WM_WINDOWPOSCHANGING



说明：窗口大小、位置或者 Z 轴层次即将改变。改变后会发送 WM_WINDOWPOSCHANGED 消息。API 函数 SetWindowPos、MoveWindow、ShowWindow 等提供相关功能。

格式：0、Integer(PWindowPos)。PWindowPos 是一个记录，包含了窗口的句柄、大小、位置、层次等信息。

WM_COPYDATA

说明：在不同窗口甚至应用程序间传递数据。

格式：HWND、Integer(PCopyDataStruct)。HWND 表示发送者的句柄，PCopyDataStruct 是一个记录，包含要传送的数据或者数据地址。

WM_NOTIFY

说明：子控件将自己的一些事件或者信息、要求告知父控件。

格式：0、Integer(PNMHdr)。PNMHdr 包含子控件句柄和告知内容。

WM_NOTIFYFORMAT

说明：子控件询问父控件在传递 WM_NOTIFY 消息时使用哪种字符集(ANSI 或者 UNICODE)。

格式：HWND、NF_QUERY/NF_REQUERY。作用过程是这样的：(1) 子控件发送(Child.Handle, NF_QUERY)；(2) 父控件返回(Parent.Handle, NF_REQUERY) 返回结果是 NFR_ANSI、NFR_UNICODE 或者 0(表示错误)。

WM_HELP

说明：通知当前激活的窗体，用户按下了 F1 键。

格式：0、Integer(PHelpInfo)。

WM_USERCHANGED

说明：当当前用户注销或者新用户登录后，系统更新用户设置后将此消息发送给所有窗体。

格式：0、0。

WM_GETICON

说明：取得窗体图标。

格式：ICON_BIG/ICON_SMALL、0。返回图标句柄。

WM_SETICON

说明：设置窗体图标。

格式：ICON_BIG/ICON_SMALL、HIcon。

WM_NCHITTEST

说明：鼠标在窗体上移动、按下或者松开。

格式：0、MakeWord(xPos, yPos)。返回鼠标所在区域。

WM_KEYDOWN

说明：非系统键(按下一个键的同时，Alt 没有按下)按下。

格式：nVirtKey、lKeyData。nVirtKey 表示按下键的虚拟代码，lKeyData 的 32 位被划分为 7 个区域，包含了一些辅助信息。

WM_KEYUP

说明：非系统键(按下一个键的同时，Alt 没有按下)松开。

格式：和 WM_KEYDOWN 同。

WM_CHAR

说明：WM_KEYDOWN 的 wParam 转化为字符得到这个消息。

格式：Integer(Char)、lKeyData。

WM_SYSKEYDOWN

说明：系统键按下。

格式：和 WM_KEYDOWN 同。

WM_SYSKEYUP

说明：系统键松开。

格式：和 WM_SYSKEYDOWN 同。

WM_SYSCHAR

说明：WM_SYSKEYDOWN 的 wParam 转化为字符得到这个消息。

格式：和 WM_CHAR 同。

WM_COMMAND

说明：菜单的某项执行、子控件给父控件发送通知消息、加速键按下。

格式：MakeWord(wID, wNotifyCode)、HWND。wID 是菜单项、控件、加速键的标识。wNotifyCode=1(如果是加速键)，=0(如果是菜单项)，别的表示通知标识码。HWND，发送通知消息时表示控件句柄，其他等于 0。

WM_SYSCOMMAND

说明：系统菜单(点窗体左上角的图标可展开它)项执行，或者点击窗体右上角的最小、最大化、关闭按钮。

格式：uCmdType、MakeWord(xPos, yPos)。uCmdType 表示要执行何种任务，xPos、yPos 表示鼠标位置。

WM_TIMER

说明：API 函数 SetTimer 可以建立一个定时器。WM_TIMER 就是定时触发的消息。

格式：wTimerID、tmprc。wTimerID 是定时器的标识，tmprc 是回调函数地址。

WM_HSCROLL

说明：控件的横向滚动条要滚动。

格式：滚动值、滚动条的句柄(如果是 Windows 标准控件，则为 0)。

WM_VSCROLL

纵向滚动。

WM_MOUSEMOVE

说明：鼠标移动。

格式：fwKeys、MakeWord(xPos, yPos)。fwKeys 指出是否同时有 Ctrl、Shift 或者鼠标左键、右键按下。

同类消息有：

WM_LBUTTONDOWN、WM_LBUTTONUP

WM_RBUTTONDOWN、WM_RBUTTONUP

WM_MBUTTONDOWN、WM_MBUTTONUP

WM_LBUTTONDBLCLK、WM_RBUTTONDBLCLK、WM_MBUTTONDBLCLK

WM_MOUSEWHEEL

等就不一一介绍了。

WM_PARENTNOTIFY

说明：子控件创建、销毁、被点击时，这个消息被发到父控件。可以参考 TComponent.Notification。

格式：MakeWord(fwEvent, idChild)、lValue。idChild 是子控件的标识码，lValue 是子控件的句柄或者鼠标位置。

WM_SIZING

说明：将改变窗口大小。

格式：fwSide、Integer(PRect)。fwSide 表示改变窗口的哪些部分(如 WMSZ_LEFT、WMSZ_TOP、WMSZ_BOTTOM 等)，第二个参数指定一个举行区域用以表示新的大小。

WM_MOVING

说明：将移动窗口位置。

格式：和 WM_SIZING 同。

WM_DROPFILES

说明：在拖放文件过程中，鼠标左键松开从而放下文件时发送。

格式：HWND、0。HWND 是关于该文件信息的数据结构的句柄，为 DragFinish、DragQueryFile、DragQueryPoint 等函数使用。

删除的内容：

WM_MOUSEHOVER、WM_MOUSELEAVE，这是 Windows 标准消息，分别在鼠标进入、离开窗口时触发。VCL 没有直接处理这两个消息，而是用 CM_MOUSEENTER 和 CM_MOUSELEAVE(定义在 Controls 单元)代替。

WM_CUT

说明：给 Edit、ComBox 发送剪切信息。

格式：0、0。

EM_UNDO、WM_CLEAR、WM_COPY、WM_PASTE 类似。

WM_USER

用户自定义消息。

Windows 消息是用数字来标识的，表 5.1 被分为五个区（见表 5.1）：

范 围	说 明
0 ~ WM_USER - 1	Windows 内部使用。WM_USER=0x0400
WM_USER ~ 0x7FFF	开发环境（如 Delphi）和软件开发人员自定义消息时使用
0x8000 ~ 0xBFFF	Windows 为将来扩展保留 其中 0xB000 ~ 0xB068 被 Delphi 定义为 VCL Control 消息
0xC000 ~ 0xFFFF	应用程序间通过自定义字符串消息通讯时使用，参看 API 函数 RegisterWindowMessage
>0xFFFF	Windows 为将来扩展保留

5.2

VCL 的消息机制

2. BM 消息

即按钮消息。它们以 BM_开头。

BM_GETCHECK

说明：获得 radio button 或者 check box 的选中状态。

格式：0、0。

BM_SETCHECK

说明：设置 radio button 或者 check box 的选中状态。

格式：fCheck、0。fCheck 可以是 BST_CHECKED/ BST_INDETERMINATE(灰色)/ BST_UNCHECKED。

BM_GETSTATE

说明：获得按钮的状态。如是否选中、是否高亮(按下)、是否具有焦点。

格式：0、0。

BM_SETSTATE

说明：获得按钮是否高亮。

格式：1/0、0。

BM_SETSTYLE

说明：改变按钮的形式。如是否有选择框、3 种状态(选中、灰色、未选中)、显示图标等。

格式：dwStyle、MakeLong(1/0, 0)。dwStyle 表示按钮形式，1/0 表示是否马上重画按钮。

BM_CLICK

说明：模拟点击按钮。它会生成 WM_LBUTTONDOWN 和 WM_LBUTTONUP 消息。类似 TControl.Click。

格式：0、0。

BM_GETIMAGE

说明：取得按钮的图片或者图标句柄(假如有)。

格式：IMAGE_BITMAP/IMAGE_ICON、0。

BM_SETIMAGE

说明：设置按钮的图片或者图标(假如能)。

格式：IMAGE_BITMAP/IMAGE_ICON、Handle。

你还可以看到 BN_开头的一些消息，严格地说它们不是消息。子控件给父控件发送通知消息时(参看 WM_COMMAND)，它们即 wNotifyCode 通知标识码。类似地还有 LBN_(用在 ListBox 中)、CBN_(用在 ComboBox 中)、EN_(用在 Edit 中)。下面就不一一介绍了，开发者一般不会使用它们。

3. CB 消息

它们用 CB_开头。相应地处理 ListBox 的消息用 LB_开头，后缀及用法相似。

TComboBox 已经封装了大多数消息，通过它的属性方法可以发送和处理这些消息。所以我们主要列出它没有封装但是有时候需要用到的 CB_消息。

CB_DIR

说明：将指定目录下符合条件的文件名和子目录名全部增加到列表中。函数 DlgDirList 和 DlgDirListComboBox 与其功能相似。

格式：uAttrs, Integer(Path)。uAttrs 指定条件。

CB_GETDROPPEDWIDTH

说明：取得下拉框的宽度。

格式：0、0。

CB_SETDROPPEDWIDTH

说明：设置下拉框的宽度。

格式：Width、0。

CB_GETTOPINDEX

说明：第一个可见项。主要在下拉框被纵向滚动后使用。

格式：0、0。

CB_SETTOPINDEX

说明：设置第一个可见项。

格式：Index、0。

4. EM 消息

用 EM_开头。同样地，被 TEdit 封装了和开发中基本不会使用的消息不作介绍，以节省篇幅。

EM_GETMODIFY

说明：内容是否已被修改。



格式：0、0。

EM_SETMODIFY

说明：设置内容是否已被修改。

格式：1/0、0。

EM_SETPASSWORDCHAR

说明：设置用户输入时显示的字符(加密字符)。

格式：Integer(Char)、0。如果第一个参数为 0，那么按输入的原样显示。

EM_GETPASSWORDCHAR

说明：得到加密字符。

格式：0、0。

EM_SETMARGINS

说明：设置显示内容时的左右边距。

格式：fwMargin、MakeLong(wLeft, wRight)。fwMargin 指定设置左边距还是右边距。

EM_GETMARGINS

说明：取得显示内容时的左右边距。

格式：0、0。返回 MakeLong(wLeft, wRight)。

EM_POSFROMCHAR

说明：取得某个字符所在位置。

格式：Integer(PPoint)、Index。Index 表示第几个字符。返回位置在第一个参数中。

EM_CHARFROMPOS

说明：取得某个位置处的字符。

格式：0、MakeLong(xPos, yPos)。返回 MakeLong(CharIndex, LineIndex)。

5. SBM 消息

它们以 SBM_开头。

SBM_SETPOS

说明：设置滚动条位置。

格式：nPos、1/0。1/0 表是否重画。

SBM_GETPOS

说明：取得滚动条位置。

格式：0、0。

SBM_SETRANGE

说明：设置滚动条滚动范围。

格式：nMinPos、nMaxPos。

SBM_SETRANGEREDRAW

说明：设置滚动条滚动范围并重画。

格式：nMinPos、nMaxPos。

SBM_GETRANGE

说明：取得滚动条滚动范围。

格式：Integer(PInteger)、Integer(PInteger)。两个参数分别表示最小和最大范围。

SBM_ENABLE_ARROWS

说明：改变滚动条的左右箭头的 Enabled 属性。

格式：fuArrowFlags、0。

SBM_SETSCROLLINFO

说明：一次设定滚动条所有参数(如 Range、Pos 等)。

格式：1/0、Integer(PScrollInfo)。1/0 表是否重画。

SBM_GETSCROLLINFO

说明：取得滚动条所有参数。

格式：0、Integer(PScrollInfo)。

6. VCL 消息

以 CM_ 开头，定义在 Controls 单元，共计 68 个。这部分是 VCL 自定义的消息（个别和 WM 标准消息相同），通用于所有类型的 VCL 组件，范围主要在 WM_USER ~ 0x7FFF。

CM_ACTIVATE

说明：窗体激活。

CM_DEACTIVATE

说明：窗体从激活变为未激活状态。

CM_GOTFOCUS

说明：窗口获得输入焦点。

CM_LOSTFOCUS

说明：窗口失去输入焦点。

注意：CM_GOTFOCUS、CM_LOSTFOCUS 在 VCL 中定义了但是没有使用。你应该用 WM_SETFOCUS、WM_KILLFOCUS 或者 CM_ENTER、CM_EXIT 代替。

CM_ENTER

说明：光标进入控件。

CM_EXIT

说明：光标离开控件。

以上消息都不需要参数，发送格式是：0、0；可用 TWMNoParams 接收。

CM_CANCELMODE

说明：告知控件取消模态。比如发给 TComboBox，那么 TComboBox 收到后应该关闭下拉框。它是 WM_CANCELMODE 的扩展。

格式：0、Integer(TControl)。可用 TCMCancelMode 接收。

CM_DIALOGKEY

说明：在 TWinControl 上按下某键后，这个消息被广播到所有子控件。常用来处理回车、Esc 等键。

CM_DIALOGCHAR

说明：在 TWinControl 上按下某字母键后，这个消息被广播到所有子控件。常用来处理加速键。

CM_FOCUSCHANGED

说明：TWinControl 上的焦点在子控件间变化时，这个消息被广播到所有子控件。

格式：0、Integer(TWinControl)。TWinControl 代表获得焦点的控件。

接收：TCMFocusChanged。

以下消息在控件自己的对应属性发生变化时被发到该控件。很明显它们都不需要参数，直接用 TMessage 接收。

CM_VISIBLECHANGED

CM_ENABLEDCHANGED

CM_COLORCHANGED

CM_FONTCHANGED

CM_CURSORCHANGED

CM_CTL3DCHANGED

CM_PARENTCTL3DCHANGED

CM_TEXTCHANGED

```

CM_MOUSEENTER
CM_MOUSELEAVE
CM_MENUCHANGED
CM_APPKEYDOWN
CM_APPSYSCOMMAND
CM_BUTTONPRESSED
CM_SHOWINGCHANGED
CM_ICONCHANGED
CM_INVOKEHELP
CM_WINDOWHOOK
CM_RELEASE
CM_SHOWHINTCHANGED
CM_PARENTSHOWHINTCHANGED
CM_SYSCOLORCHANGE
CM_WININICHANGE
CM_FONTCHANGE
CM_TIMECHANGE
CM_TABSTOPCHANGED
CM_UIACTIVATE
CM_UIDEACTIVATE
CM_DOCWINDOWACTIVATE
CM_GETDATA LINK
CM_DIALOGHANDLE
CM_ISTOOLCONTROL
CM_RECREATEWND
CM_INVALIDATE
CM_SYSFONTCHANGED
CM_ISSHORTCUT
CM_BORDERCHANGED
CM_BIDIMODECHANGED
CM_PARENTBIDIMODECHANGED
CM_PARENTFONTCHANGED
CM_PARENTCOLORCHANGED
CM_ALLCHILDRENFLIPPED
CM_ACTIONUPDATE
CM_ACTIONEXECUTE

```

```
CM_CHANGED
```

说明：当子控件的属性发生了变化，并且这些变化可能影响到父控件时，应该发送这个消息给父控件。可以用 TControl.Changed 发送。



格式：0、Integer(TControl)。

接收：TCMChanged。

CM_HINTSHOWPAUSE

说明：是否显示 Hit，并指定显示的时间。

格式：1/0、时间。

接收：TCMHintShowPause。

CM_DOCKCLIENT

说明：入坞消息。它在 TWinControl.OnDockDrop 前被发送。

格式：Integer(Source)、Integer(TSmallPoint)。

接收：TCMDockClient。

CM_UNDOCKCLIENT

说明：离坞消息。它在 TWinControl.OnUnDock 后被发送。

格式：Integer(NewTarget)、Integer(Client)。

接收：TCMUnDockClient。

CM_DRAG

说明：拖放消息。

格式：TDragMessage、Integer(PDragRec)。

接收：TCMDrag。

CM_HINTSHOW

说明：显示 Hint。

格式：0、Integer(PHintInfo)。

接收：TCMHintShow。

CM_MOUSEWHEEL

说明：和 WM_MOUSEWHEEL 相似。

接收：TCMMouseWheel。

另外，一些 VCL 组件也定义了自己的消息(位于 WM_USER ~ 0x7FFF 区，以组件类名简写开头)，开发人员一般不需要直接使用它们，这里就不介绍了。

附注：消息数据类型

消息的数据(包括消息代号、两个参数和返回值)是一个记录类型的变量。典型的消息数据类型是 TMessage，它可以容纳任何消息类型的数据。

下面是 VCL 中对 TMessage 的定义(Messages 单元)：

```
type
  TMessage = packed record
    Msg: Cardinal;
    Case Integer of
      0: (
        WParam: Longint; {Pascal 中的 Longint 和 Integer 是相同的类型}
        LParam: Longint;
        Result: Longint);
      1: (
        WParamLo: Word;
        WParamHi: Word;
        LParamLo: Word;
        LParamHi: Word;
        ResultLo: Word;
        ResultHi: Word);
    end;
```

Msg 是消息代号，WParam、LParam 是消息参数。后面加 Lo 或 Hi，表示低 16 位或者高 16 位数据。这是一个典型的变体记录 (record with variant parts)，关于变体记录更详细的情况参见 3.1.4。

在 Windows 和 VCL 消息数据(WParam、LParam 和 Result)中，高 16 位和低 16 位有时候分别表示不同的意思，而整个的数据可能反而没有实际意义，所以 Delphi 中定义了一个变体记录，方便用户存取它们整体或者高 16 位和低 16 位数据。

小结

本小节详细介绍了 Windows 标准消息和 VCL 自定义消息的分类和常用消息的含义、使用方法。

消息分类和常用消息的使用方法是本节的重点。本小节的内容也可以作为编写程序时选择消息的参考资料。

5.3 多态性

多态性，是一个很基本但又非常容易让人误解的概念。因为它是个很基本的东西，所以也常常被人遗忘，尽管我们总是在使用它。

比如下面的代码：

```
var
  Button: TButton;
begin
```

5.3

多态性


```

    Button := TButton.Create(Self);
    .....
    Button.Free;
end;

```

在上面的“Button.Free”中，已经利用了多态，你是否曾注意了昵？

多态性，并不是 VCL 的特性，它应该被看做是语言的特性。但是它又不仅仅是 Object Pascal 的特性，因为很多语言都有多态性。之所以将多态性放在这章的最后一节来讲，是因为要弄清多态性这个很基本的概念，却要牵涉到上面章节讲过的很多基本知识，如类成员的重新声明、虚方法、覆盖、重载等。

5.3.1 多态性的概念

很通俗地讲，多态性是指可以将子对象赋值给父对象的技术，可以通过父对象调用子对象从父对象继承并覆盖了的虚方法。或者说，多态性是可以让父对象具有不同行动方式的技术。

注意上面一句话包含了两个特性：

- (1) 可以将子对象赋值给父对象的技术。
- (2) 可以通过父对象调用子对象从父对象继承并覆盖了的虚方法。通过父对象调用子对象的虚方法时，执行的是子对象覆盖后的代码，而不是父对象实现该虚方法的代码。

我们看下面一段代码：

```

unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
    Forms, Dialogs, StdCtrls;

type
    TForm1 = class(TForm)
        Button1: TButton;
        Button2: TButton;
        procedure Button1Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

```

end;

end;

end;

```
end;
```

{ 第一个特性：可以将子对象赋值给父对象的技术 }

删除的内容: ,

5.3

多态性

```

ObjP := ObjC;
{ 第二个特性：可以通过父对象调用子对象从父对象继承并覆盖了的虚方法。执行
  ObjP.Method1 后，显示的是"TChild"而不是"TParent" }
ObjP.Method1;
{ 下面一句也体现了第二个特性 ObjP.Free 实际执行的是TChild的Free 而不是TParent
  的Free。因为Free是TObject的一个虚方法 }
ObjP.Free;
end;

end.

```

5.3.2 多态性和虚方法的关系

可以这么说，多态性是体现在虚方法上的，虚方法及其覆盖是实现多态性的手段。

多态性的概念对于其他类成员或者非虚方法则是不适用的。这可以分为两方面来说：

(1) 对于子对象自己声明的而不是从父对象继承的成员，通过多态是不可能访问这些成员的，编译都不能通过。这也说明，多态本质上是一种运行时特性。看如下的代码：

```

procedure TForm1.Button3Click(Sender: TObject);
var
  Obj: TObject;
begin
  Obj := Self;
  {Self 是一个TForm类型的对象，具有Tag 属性；但是通过Obj 来访问Tag 的企图则是不
    能达到的。下面一句不能通过编译}
  ShowMessage(IntToStr(Obj.Tag));
end;

```

(2) 多态性也不适用于重新声明后的类成员。

我们看下面的代码：

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type

```

```
procedure TParent.Method1;  
begin  
    ShowMessage( 'TParent' );  
end;
```



```
{ TChild }  
procedure TChild.Method1;  
begin  
    ShowMessage('TChild');  
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
var  
    ObjP: TParent;  
    ObjC: TChild;  
begin  
    ObjC := TChild.Create;  
    ObjP := ObjC;  
    ObjC.StrProp := 'TChild';  
    { Method1 显示"TParent" 而不是"TChild" }  
    ObjP.Method1;  
    { StrProp 为空而不是"TChild" }  
    ShowMessage(ObjP.StrProp);  
    ObjP.Free;  
end;  
  
end.
```

至于多态性是怎么通过虚方法来实现的，可以参考“虚拟方法表和动态方法表”小节。

最后要说明一点的是，函数和过程重载并非属于我们上面所讲多态概念的范畴，尽管它们在“具有不同行动方式”这一点上有类似之处。重载更应该被看做是另一种语言特性，和多态并没有直接关系。

第 6 章 组件开发实战

学习开发 VCL 组件,是提高 Delphi 开发水平的重要途径。在本书前几章里已经学习了 Object Pascal 的重要基础知识,并熟悉了 VCL 架构体系。在本章,是放下菜谱,抄刀捉勺进行实战的时候了。

本章将通过多个组件开发实例来阐述组件开发的三种方法,其中穿插了组件开发的许多实用技巧和注意事项。开发的组件涉及到不可视组件、数据敏感控件、图形图像控件和 QuickReport 报表组件等。

写作本章颇费了一番脑筋。我将这些组件开发出来后,已在工作中使用很长时间了,因此,费脑筋的不是组件开发本身的技术问题,而是怎么用浅显易懂的语言来描述组件原理和开发过程。斗争了很长时间,最后决定,对于复杂一些的、功能多一些的组件,大致用三步来讲述其开发过程:

- (1) 说明原理。主要是用描述性的文字。
- (2) 在应用程序中用代码简要说明开发过程的关键部分。而不是上来就搞什么类继承、组件派生。
- (3) 组件封装。整理上几个步骤的代码并做一些善后工作,从而得到组件的完整源代码。

本章实例中开发的组件都是完整的,也就是可以直接在实际软件开发中使用。所有组件的源代码及源代码注释都保存在本书的附带光盘中。

6.1 三种组件开发方法

我国北魏时人贾思勰写了一部农学巨著:《齐民要术》。该书“起自耕农,终于区自(意思是饮食)”,全书九十二篇,有二十五篇涉及饮食烹饪。这二十五篇全面总结了以往的烹饪方法:煮、煎、炸、炙、烩、绿(有学者以为是熘)。可见烹饪的方法是有章可循的。

组件开发也一样,并不是复杂得让人摸不着头脑,不知道从哪里下手。这里所说的组件开发方法是从 VCL 架构角度而言,并不是指组件的具体开发方法。通观整个 VCL,我们发现它至少使用了以下三种方法来构建整个架构:

- (1) 继承;(2) 聚合;(3) 子类化。

熟练掌握以上方法,将帮助你设计出符合 VCL 标准架构、利于将来扩展和组织组件。

6.1.1 继承、聚合和子类化

1. 继承

所有的 VCL 组件都不是无源之水,它们总是有特定的父类。换句话说,一个类总是继承了父类的特性,并添加和修改部分特性,从而成为一个新的组件。

2. 聚合

由多个类合成一个组件。但是多个类中必须只能有一个成为主类，其他成为辅助类。如 TRadioGroup 是由 TCustomGroupBox 和 TGroupButton(TRadioButton 的子类)聚合而成的，TLabelEdit 是由 TCustomEdit 和 TBoundLabel 聚合而成的。其中 TCustomGroupBox 和 TCustomEdit 是主类。

在聚合组件中，如果有必要，可以将辅助对象作为组件的属性发布(如很典型的 DataSource 属性)，否则可以作为私有变量隐藏起来，而提供一些属性和方法来实现对它的访问。

3. 子类化

当我们要开发大量同类组件时，首先需要开发一个高端类，然后从高端类派生子组件。高端类应该尽可能声明子组件的共性成员，如果成员的实现依赖于具体子类，那么可以声明为虚或者抽象成员，目的是可以使用相同方法访问共性成员。VCL 中包含了大量的 'Custom' 开头的类，如 TCustomControl、TCustomEdit 等，就是这里所说的高端类。子类化和继承的概念有交叉部分，但是它们是从不同角度来说的。子类化侧重的是 VCL 的整个架构规划。

在本章里，我们主要采用继承和聚合的方法来开发组件，从而演示开发组件的常用方法和使用技巧。使用继承方法时，重点是选择最佳的父类。

(1) 要保证父类是最轻捷的，避免给用户提供不必要的属性和方法，也就是说，尽可能选择 VCL 架构中更顶层的类作为父类。在满足要求的前提下，宁选择 TComponent 而不选择 TControl，宁选择 TControl 而不选择 TWinControl。

比如，开发一个下面功能的组件：它有一个 DataSet:TDateSet 属性，要求根据 DataSet 的字段信息，自动在界面上生成每个字段对应的数据敏感控件。显然，这个组件主要提供一个 DataSet 属性和一个生成敏感控件的方法，控件本身是不需要可见的。那么我们选择 TComponent 作为父类就足够了，而没必要选择 TControl 等搞出一大堆不需要的属性和方法。

(2) 要选择尽可能贴近新组件功能要求的父类，这样可以减少开发工作量。我们没必要重新开发 VCL 组件已经具有的功能，何况一般情况下也没它写得好！比如开发图形图像控件时，选择 TGraphicControl 或其子类作为父类，而没必要选择 TControl 来重新实现一个 TCanvas。

(3) 以上两点在某些时候是彼此矛盾的，此时就要善于取舍，抓住主要矛盾、兼顾次要矛盾，然后找到其平衡点，双方都要作出牺牲，但最终保证损失最小化。

采用聚合方法开发组件时，要注意以下几点：

(1) 主类和辅助类的父类都宜按照上面所讲继承法的原则进行选择。

(2) 要建立将辅助对象的销毁信息通知主对象的机制。具体做法是：

首先，调用辅助对象的 FreeNotification(AComponent: TComponent)方法注册辅助对象（参数 AComponent 是主对象）的销毁通知。这样，辅助对象被销毁时，会及时通知主对象，以便主对象将辅助对象变量置为 nil，避免产生野指针（也称悬挂指针）。

然后，覆盖主类的 Notification 虚方法，在得到销毁消息时及时将辅助对象变量置为 nil。

需要说明的是，如果主对象和辅助对象的 Owner 相同，那么 VCL 会自动通知，不需要人工调用

FreeNotification；否则需要。在不清楚它们的 Owner 是否相同或者不想判断时，也应该人工调用 FreeNotification，它在内部会自动判断。

(3) 主对象和辅助对象要保持互动。如主对象的 Enabled、位置等发生变化时，辅助对象应该作响应调整。

(4) 如果辅助类要实现众多加强功能，那么可以单独定义为一个新类，在新类里实现这些功能。这样可以使组件的条理更加明晰，提高可读性。

6.1.2 接口、虚方法和辅助类的选择

VCL 中很少使用接口。在 Delphi 中，接口除了用在 COM 中（也是它主要而必须使用的地方），就是解决多继承的问题（VCL 中不能一次从多个类派生一个子类，而只能单根继承）。理论上接口可以解决任何多继承的问题，但是凡事要考虑效率，力求简便。根据接口只声明无实现的特点中知道，它只适合于大量子类有很多同类方法，但是具体实现相差很大的情况。

表 6-1 说明了 Delphi 中几种典型的类似多继承问题及其解决方法。

表 6-1 Delphi 中几种典型的类似多继承问题及解决方法

问题描述	解决方法
大量类有很多同类方法，但是具体实现相差很大	接口
父类方法的具体实现完全或者部分依赖子类	虚方法
大量类要求有类似功能，而具体实现差别不大或者没有差别	辅助类

小结

本节介绍了组件开发的三种方法：继承、聚合、子类化。同时讨论了接口、虚方法和辅助类在组件开发中的选择问题。

6.2 文件拖放监视器

我们在使用 Windows Media Player 和 Real Player 等播放器时，从 Windows 资源管理器中选择多个文件并拖动到播放器中，播放器就可以播放这些文件。这到底是怎么回事呢？

6.2.1 文件拖放原理

原来这里用到了一个很重要的 Windows 消息：

WM_DROPFILES



在 Windows SDK 中，对消息 WM_DROPFILES 是这样解释的：

一个应用程序中的控件被注册为文件（或者文件夹，以下统称文件）拖放接收者后，文件被拖动并放到该控件（即鼠标左键在控件上被松开）时，控件会收到 WM_DROPFILES 消息。

一个完整的文件拖放过程是这样的：

(1) 首先使用 API 函数：

```
procedure DragAcceptFiles(Wnd: HWND; Accept: BOOL); stdcall;
```

将一个控件（由 Wnd 指定）注册，以便它可以接收文件拖放消息 WM_DROPFILES。

(2) 文件被拖放到该控件时，控件会收到 WM_DROPFILES 消息。这时候就可以调用另一个 API 函数：DragQueryFile。通过它可以获得拖放信息，如：文件个数、文件路径甚至鼠标松开时所处的位置等。

(3) 当不再需要文件拖放消息时，再次调用 DragAcceptFiles 注销该控件（此时给参数 Accept 传入 False 即可）。

很简单吧，你查一下 SDK 就可以自己动手了！

6.2.2 文件拖放实例

打开 Delphi，选择菜单 File|New|Application，这样就新建了一个工程。然后拖一个 TListBox 到窗体 Form1 上，用它来显示拖放到 Form1 的文件的路径。

然后根据上小节的步骤可以轻易地写出如下的代码：

```
.....
type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  protected
    procedure WndProc(var Message: TMessage); override;
  end;

var
  Form1: TForm1;

implementation

uses ShellAPI;
{ DragAcceptFiles 和 DragQueryFile 是声明在 ShellAPI 单元的}

{$R *.dfm}
```

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    DragAcceptFiles(Handle, True);    { 将Form1 注册为文件拖放接收控件 }
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    DragAcceptFiles(Handle, False);   { 注销Form1 }
end;

{ 覆盖Form1 的窗口过程WndProc, 这样就可以捕获WM_DROPFILES 消息 }
procedure TForm1.WndProc(var Message: TMessage);
var
    Count, Index, hDrop: Integer;
    PFileName: PChar;
begin
    if Message.Msg = WM_DROPFILES then
    begin
        hDrop := Message.WParam;    { 取得系统drop 结构的句柄, 在后面要用到它 }
        GetMem(PFileName, MAX_PATH);
        { 取得文件个数 }
        Count := DragQueryFile(hDrop, MAXDWORD, PFileName, MAX_PATH-1)
        ListBox1.Items.Clear;
        for Index := 0 to Count-1 do
        begin
            DragQueryFile(hDrop, Index, PFileName, MAXBYTE);
            { 取得每个文件的路径并放入缓冲区 PFileName }
            ListBox1.Items.Add(PFileName);
        end;
        FreeMem(PFileName);
        DragFinish(hDrop);
        { 当WM_DROPFILES 被处理后, 应该调用DragFinish 释放资源 }
    end else
        inherited;
    end;

```

然后按 F9 键就可以运行程序了。你可以拖几个文件到 Form1 上,看看有什么结果。你可能只有一个感觉,那就是:太简单了!

6.2.3 组件封装

如果就此满足当然是不行的,因为上面的代码将 Form1 固定为文件拖放接收者,将 ListBox1 固定





为文件信息显示者了。当我们要改变这二者时，虽然可以简单地 Copy&Paste 后修改几处即可，但这无疑是新手和性子急躁者的行事方法。

其实，将上面的代码封装为组件，也是极其容易的。我们发布一个属性，代表文件拖放接收者；再发布一个事件，反馈文件信息。

我们可以将这个组件称为文件拖放监视器。它可以如下定义：

```
type
  TDropFilesEvent = procedure(Receiver: TWinControl;
    const FileNames: TStrings; const FilesCount: Integer;
    const DropPoint: TPoint) of object;

  TlxpDraggingFilesMonitor = class(TComponent)
  { 由于组件在运行时不需要可见，所以从TComponent派生 }
  private
    FAcceptFilesControl: TWinControl; { 指定接收者 }
    OldWindowProc: TWndMethod; { 保存FAcceptFilesControl原来的窗口过程 }
    FFileName: TStrings; { 拖放的文件名列表 }
    FOnDropFiles: TDropFilesEvent; { 拖放事件 }
    procedure SetAcceptFilesControl(const Value: TWinControl);
  protected
    { FAcceptFilesControl新的窗口过程 }
    procedure NewWindowProc(var Message: TMessage);
    procedure Notification(
      AComponent: TComponent; Operation: TOperation); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property AcceptFilesControl: TWinControl read FAcceptFilesControl
      write SetAcceptFilesControl;
    property OnDropFiles: TDropFilesEvent read FOnDropFiles write
      FOnDropFiles;
  end;
```

其中：

(1) TDropFilesEvent 是一个对象过程类型。其参数：

Receiver 表示哪个控件接收拖放的文件；FileNames 是所有被拖放的文件的文件名(含路径)；Count 是文件数目，即 FileNames.Count；DropPoint 是鼠标在接收控件上的松开位置。

(2) OldWindowProc 和 NewWindowProc 合作完成 AcceptFilesControl 的消息处理。

现在来看看实现：

6.2 文件拖放监视器

```

Count, Index, hDrop: Integer;
PFileName: PChar;
P: TPoint;
begin
  if Message.Msg = WM_DROPFILES then { 如果是文件拖放消息 }
  begin
    hDrop := Message.WParam;
    FFileName.Clear;
    GetMem(PFileName, MAX_PATH); { 给缓冲区 PFileName 分配空间 }
    Count := DragQueryFile(hDrop, MAXDWORD, PFileName, MAX_PATH-1);
    { 取得文件总个数 }
    for Index := 0 to Count-1 do
    begin
      DragQueryFile(hDrop, Index, PFileName, MAXBYTE);
      { 得到每个文件的名字 }
      FFileName.Add(PFileName);
    end;
    DragQueryPoint(hDrop, P); { 取得鼠标松开时的位置, 保存到 P }

    if Assigned(FOnDropFiles) then { 如果用户书写了事件代码, 则触发事件 }
      FOnDropFiles(FAcceptFilesControl, FFileName, Count, P);

    FreeMem(PFileName); { 释放缓冲区空间 }
    DragFinish(hDrop); { 完成本次拖放 }
  end else
    OldWindowProc(Message); { 调用原来的 WindowProc 过程处理别的消息 }
end;

procedure TlxpDraggingFilesMonitor.Notification(
  AComponent: TComponent; Operation: TOperation);
begin
  inherited;
  if (AComponent = FAcceptFilesControl) and (Operation = opRemove)
  then
    FAcceptFilesControl := nil;
end;

```

接下来我们可以测试这个组件：

新建一个工程，放置一个 TListBox 和一个 TlxpDraggingFilesMonitor 到 Form1 上，设置 lxpDraggingFilesMonitor1 的 AcceptFilesControl 为 ListBox1。双击 lxpDraggingFilesMonitor1 在事件

lxdraggingFilesMonitor1DropFiles 中写如下代码：

```
TListBox(Receiver).Items.Assign(FileNames);
```

然后运行程序，打开资源管理器，用鼠标选择一些文件和文件夹，拖入 Form1.ListBox1，松开鼠标。怎么样，所有文件和文件夹的路径和名字被加到了 ListBox1 中。

小结

本节的重点是：

- (1) 如何发布新的事件以及新事件的触发。
- (2) 如何处理辅助对象的消息。
- (3) 辅助对象指针及时置 nil。

6.3 托盘组件

一个应用程序运行后，可以在 Windows 任务栏放入一个图标，这个图标叫做托盘图标。用鼠标点击该图标时，可以执行一些操作。比如双击时，将最小化的应用程序正常显示；点击右键时，可以弹出一个菜单，执行“打开、退出”等功能。

在本节里，我们要开发一个托盘组件。这个组件的功能是：

(1) 在它所属应用程序运行时，将一个图标放入托盘区；图标可以由用户指定，如果没有指定，则默认使用应用程序的图标。

- (2) 可以由用户指定一个右键弹出菜单。
- (3) 可以显示动画图标，即交替显示无图像图标和有图像图标。和 QQ 接收消息时的动画类似。
- (4) 可以由用户指定当有用用户登录系统时，是否自动启动应用程序。

以上这些功能比较复杂，为了便于理解，我们拆分为以下几个步骤：

- (1) 装入托盘图标。
- (2) 在应用程序最小化时去掉状态栏的图标，因为我们已经有了托盘图标。
- (3) 给托盘图标增加接收鼠标消息功能。
- (4) 处理鼠标消息。
- (5) 显示动画图标。
- (6) 设置程序的自动启动功能。

我们先用不同的代码片断逐步实现以上功能，最后封装为一个组件。

6.3.1 装入托盘图标

一个托盘图标，在内部实际上是如下一个记录体：

```
TNotifyIconData = record
```



托盘组件

```

cbSize: DWORD;
Wnd: HWND;
uID: UINT;
uFlags: UINT;
uCallbackMessage: UINT;
hIcon: HICON;
szTip: array [0..63] of AnsiChar;
end;

```

字段含义如下：

cbSize：记录体的大小。我们知道调用 API 函数时，字符串和记录体参数都是通过指针或者地址传送的。对于一个字符串 PChar，因为它以零字符（#0/NULL）结束，所以 API 函数可以根据零字符确定字符串的长度；但是一个记录可以定义任意个数的字段，因此通常需要传送一个记录体大小给 API 函数，否则 API 函数无法知道记录体的结束位置。

Wnd：托盘图标窗口的句柄。托盘图标需要内建一个窗口（可以是不可见的），并创建一个关联的窗口过程，否则无法接收消息。如果没有一个内建窗口和关联的窗口过程，鼠标移动到图标点击时就不会有什么事件发生，因为图标无法接收到这些消息。

uID：托盘图标代号。

uFlags：说明 TNotifyIconData 的哪些字段值是有效的。可以是以下三个取值之一或者组合：

NIF_ICON	hIcon 有效
NIF_MESSAGE	uCallbackMessage 有效
NIF_TIP	szTip 有效

uCallbackMessage：通知消息的代号。当鼠标移动到和点击图标时，系统通知相应的消息给窗口，消息类型和信息在 TMessage 记录体中指定。

hIcon：图标的句柄。

szTip：托盘图标的提示信息。

这个记录体是供下面一个 API 函数使用的：

```

function Shell_NotifyIcon(dwMessage: DWORD; lpData: NotifyIconData):
    BOOL; stdcall;

```

Shell_NotifyIcon 给系统托盘区发送消息，从而修改托盘图标。该函数定义在 ShellAPI 单元，参数含义如下：

dwMessage：可以是以下三个值之一：

NIM_ADD	增加一个图标
NIM_DELETE	删除一个图标
NIM_MODIFY	修改一个图标

lpData：就是上面所讲 TNotifyIconData 记录体的指针。

下面看一段代码，这段代码可以添加一个图标到系统托盘区：

```
uses ShellAPI;

var
  FIconData: TNotifyIconData;
  FIcon: TIcon;
begin
  FIcon := TIcon.Create;
  { 从 ImageList1 提取一个图标。事先应该在 ImageList1 添加一些 .ico 文件以装入图标 }
  ImageList1.GetIcon(0, FIcon);

  { 清零 FIconData }
  FillChar(FIconData, SizeOf(FIconData), #0);
  with FIconData do
  begin
    { 指定记录的大小 }
    cbSize := SizeOf(FIconData);
    { 使参数 hIcon 有效，因为这个例子没有建立消息窗口和窗口过程，也没有设置图标提示信息，所以参数 hIcon 有效 }
    uFlags := NIF_ICON;
    { 指定图标句柄 }
    hIcon := FIcon.Handle;
  end;
  { 向托盘区添加图标 }
  Shell_NotifyIcon(NIM_ADD, @ FIconData);

  FIcon.Free;
end;
```



运行以上一段代码，托盘区的确增加了一个图标！不过当你将鼠标移到图标上时，图标就消失了，这是因为没有给图标指定窗口和窗口过程。

6.3.2 在应用程序最小化时去掉状态栏的图

首先需要截获应用程序最小化的消息。回顾我们讲过的八种处理消息的方法，也许我们首先想到的是使用 Application.OnMessage。没错，用它的确可以达到目的。但是君子取之有道，因为我们讲过了 Application.OnMessage 的重大弊端，所以还是摒弃之。

应用程序最小化时，主窗体也会最小化，所以还是在主窗体最小化上打主意吧。对于八种处理消

息的方法，我们有过总结：“事件嫁接是一种很值得重视的编程技巧。嫁接的原理对于所有可见方法也是适用的。如果希望在一个类中处理另一个类的消息，嫁接是一种极好的选择”。

的确，我们在这里又要用到这个技巧了，那就是嫁接主窗口的属性 WindowProc。请看以下代码：

```
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    OldWindowProc: TWndMethod;
    procedure NewWindowProc(var Message: TMessage);
  end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  {用OldWindowProc将WindowProc保存}
  OldWindowProc := WindowProc;
  {给WindowProc指定新的值NewWindowProc}
  WindowProc := NewWindowProc;
end;

procedure TForm1.NewWindowProc(var Message: TMessage);
begin
  if Assigned(OldWindowProc) then
    OldWindowProc(Message);
  with Message do
    if (Msg = WM_SYSCOMMAND) and
      (WParam = SC_MINIMIZE) then {如果用鼠标点击窗口的最小化按钮，则隐藏窗}
    □}
    ShowWindow(Application.Handle, SW_HIDE);
end;
```

运行以上代码，最小化程序后再看看状态栏，如何，的确没有 Application 的窗口了！

注意在最后封装组件时，TForm1.FormCreate 中的代码会作重大修改。也许你认为就是修改成下面这个样子：

```
OldWindowProc := Application.MainForm.WindowProc;
Application.MainForm.WindowProc := NewWindowProc;
```

看起来似乎没有任何问题，但如果你这样修改后运行程序，会发生致命异常！这是为什么呢？

原来这时 `Application.MainForm = nil`！`Application` 的属性 `MainForm` 还没有被初始化。
正确的代码请参看本节最后的源代码。

6.3.3 给托盘图标增加接收鼠标消息功能

增加接收鼠标消息功能的工作就是要给图标内建窗口，并建立窗口过程。

```
uses
    ShellAPI;

type
    TForm1 = class(TForm)
        Button1: TButton;
        ImageList1: TImageList;
        procedure Button1Click(Sender: TObject);
        procedure FormCreate(Sender: TObject);
        procedure FormDestroy(Sender: TObject);
    private
        FIconData: TNotifyIconData;
        {FNotificationWnd 用来保存内建窗口的句柄}
        FNotificationWnd: HWND;
        {FHint 用来保存提示信息}
        FHint: String;
        FIcon: TIcon;
        {定义一个窗口过程，它和FNotificationWnd 关联}
        procedure NotificationWndProc(var Message: TMessage);
    end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
    FIcon := TIcon.Create;
    ImageList1.GetIcon(0, FIcon); {取得 ImageList1 的第一个图标并放入 FIcon}

    {调用 Classes 单元的全局函数 AllocateHWND 创建一个窗口，该窗口的窗口过程是
```



```

    NotificationWndProc。函数返回窗口句柄，存入 FNotificationWnd}
    FNotificationWnd := Classes.AllocateHWnd(NotificationWndProc);

    { 指定提示信息 }
    FHint := 'lxpbuaa';
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    { 销毁内建窗口 }
    if FNotificationWnd <> 0 then
        Classes.DeallocateHWnd(FNotificationWnd);
    FIcon.Free;
end;

procedure TForm1.NotificationWndProc(var Message: TMessage);
begin
    { 我们暂时只简单调用 API 函数 DefWindowProc 作缺省处理 }
    Message.Result := DefWindowProc(
        FNotificationWnd, Message.Msg, Message.WParam, Message.LParam);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    FillChar(FIconData, SizeOf(FIconData), #0);
    with FIconData do
    begin
        cbSize := SizeOf(FIconData);
        { 重新设置有效参数 }
        uFlags := NIF_MESSAGE or NIF_ICON or NIF_TIP;
        { 指定接收消息的窗口句柄 }
        Wnd := FNotificationWnd;
        hIcon := FIcon.Handle;
        { 写入提示信息 }
        StrLCopy(szTip, PChar(FHint), SizeOf(szTip));
    end;
    Shell_NotifyIcon(NIM_ADD, @FIconData);
end;

```

带格式的

运行以上代码，我们发现鼠标遇到托盘图标时，图标再也不会消失了，反而还跳出一个提示信息！

如图 6-1 所示。



图 6-1 托盘图标

6.3.4 处理鼠标消息

处理鼠标消息主要分为两部分：

- (1) 双击时，正常显示应用程序。
- (2) 点击右键时，弹出菜单。

因此有如下代码：

```
const
    ICON_ID = 1;
    MI_ICONEVENT = WM_USER + 1;
    .....
```

在 Button1Click 中加上：

```
uID := ICON_ID;
uCallbackMessage := MI_ICONEVENT;

{ 处理消息 }
procedure TForm1.NotificationWndProc(var Message: TMessage);
var
    P: TPoint;
begin
    if Message.Msg = MI_ICONEVENT then
    begin
        case Message.LParam of
            { 如果用户双击，则正常显示程序 }
            WM_LBUTTONDBLCLK:
                RestoreApp;
            { 如果点击右键，则在点击处显示弹出菜单 }
            WM_RBUTTONDOWN:
                begin
                    GetCursorPos(P);
                    PopupMenu1.Popup(P.X, P.Y);
                end;
        end;
    end;
```



```

    end else Message.Result :=
        DefWindowProc(FNotificationWnd, Message.Msg, Message.WParam,
            Message.lParam);
    end;

    { 过程RestoreApp用来正常显示程序 }
    procedure TForm1.RestoreApp;
    begin
        { 显示应用程序窗口 (即状态栏的那个条) }
        ShowWindow(Application.Handle, SW_SHOWNORMAL);
        { 显示应用程序的主窗口 }
        ShowWindow(Application.MainForm.Handle, SW_SHOWNORMAL);
        { 将该应用程序主窗口设置到当前显示的所有窗口的最顶层 }
        SetForegroundWindow(Application.MainForm.Handle);
    end;

```

运行改过的程序，的确是按我们的意愿执行的，很好。那么可以进入下一步了。

6.3.5 显示动画图标

显示动画图标的原理，就是在不同的时间间隔点改变图标。更详细地讲，就是改变结构 TNotifyIconData 的 hIcon 参数，然后用 Shell_NotifyIcon(NIM_MODIFY, @FIconData)送出。

因此，在上面代码的基础上，我们在一个 TTimer 的事件中写如下代码就可以了：

```

procedure TForm1.Timer1Timer(Sender: TObject);
begin
    with FIconData do
    begin
        { 交替显示图标和无图像的图标 }
        if hIcon <> 0 then
            hIcon := 0 { 显示无图像图标 }
        else
            hIcon := FIcon.Handle; { 显示原来的图标 }
        end;
        Shell_NotifyIcon(NIM_MODIFY, @FIconData);
    end;
end;

```

6.3.6 设置程序的自动启动功能

这个功能的实现十分容易，就是在注册表项：

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
```

下添加一项就可以了：

```
uses Registry;

var
  Reg: TRegistry;
  KeyName: String;
begin
  Reg := TRegistry.Create;
  {ParamStr(0) 等价于 Application.ExeName ;但是使用 Application 变量需要 Forms
  单元的支持。在开发组件时，当然是引用的单元越少越好}
  KeyName := ExtractFileName(ParamStr(0));
  with Reg do
  try
    RootKey := HKEY_LOCAL_MACHINE;
    if OpenKey('\Software\Microsoft\Windows\CurrentVersion\Run', False)
    then
    begin
      { 以应用程序名为键名、应用程序全路径（含应用程序名）为键值写注册表项 }
      WriteString(KeyName, ParamStr(0));
      CloseKey;
    end;
  finally
    FreeAndNil(Reg);
  end;
end;
```



6.3.7 组件封装

封装过程几乎不用讲了，惟一要解释的是动画图标的实现。

记得在上面实现这个功能时，采用了 TTimer 组件，不过这里要封装为一个组件，所以使用了一个 API 函数 SetTimer 来设置定时器。并公开了一个方法：

```
procedure ShowTrayIcon(
  Mode: Cardinal = NIM_ADD; Animated: Boolean= False);
```

在需要动画显示图标时，按下面格式调用：

```
ShowTrayIcon (NIM_MODIFY, True);
```

即可。要停止动画时，调用：

```
ShowTrayIcon (NIM_MODIFY, False);
```

或者：

```
ShowTrayIcon (NIM_MODIFY);
```

下面列出组件的完整代码：

```
unit lxpTrayIcon;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Menus, ShellApi,
  ExtCtrls;

const
  ICON_ID = 1;
  MI_ICONEVENT = WM_USER + 1;      { 自定义一个消息 }

type
  TlxpTrayIcon = class(TComponent)
  private
    FTrayIcon: TIcon;
    FInterval: Cardinal;
    FPopupMenu: TPopupMenu;
    FNotificationWnd: HWND;
    FHint: String;
    FStartAtBoot: Boolean;
    FOnDblClick: TNotifyEvent;
    TimerHandle: LongWord;
    NotifyIconData: TNotifyIconData;
    OldWindowProc: TWndMethod;
    procedure NotificationWndProc(var Message: TMessage);
    procedure SetTrayIcon(const Value: TIcon);
    procedure SetStartAtBoot(const Value: Boolean);
    procedure Registry(B: Boolean);
    procedure NewWindowProc(var Message: TMessage);
  protected
    procedure DoDblClick;
    procedure Notification(AComponent: TComponent; Operation: TOperation);
  override;
```

{Loaded 是 TComponent 的一个虚拟方法。当所有组件被创建，并从 dfm 文件读出数据初始化这些组件实例后，Loaded 方法被自动调用。在 Loaded 中可以进行额外的初始化工作，可以对组件实例的一些成员进行改变、嫁接}

```

procedure Loaded; override;
public
    constructor Create(AOwner: TComponent);override;
    destructor Destroy;override;
    {应用程序中可以直接调用组件的 RestoreApp 方法来正常显示应用程序}
    procedure RestoreApp;
    {应用程序中可以直接调用组件的 ShowTrayIcon 方法来正常托盘图标}
    procedure ShowTrayIcon(
        Mode: Cardinal = NIM_ADD; Animated: Boolean = False);
published
    {属性 Hint 表示托盘图标的提示信息}
    property Hint: String read FHint write FHint;
    {用户双击托盘图标时激发 OnDblClick 事件}
    property OnDblClick: TNotifyEvent read FOnDblClick write FOnDblClick;
    {用户右键单击托盘图标时弹出菜单 PopupMenu}
    property PopupMenu: TPopupMenu read FPopupMenu write FPopupMenu;
    {属性 TrayIcon 让用户自己设置托盘图标}
    property TrayIcon: TIcon read FTrayIcon write SetTrayIcon;
    {属性 StartAtBoot 让用户设置应用程序是否在有用户登录系统时自动启动}
    property StartAtBoot: Boolean read FStartAtBoot write SetStartAtBoot;
    {属性 Interval 表示动画图标交替显示的时间间隔}
    property Interval: Cardinal read FInterval write FInterval;
end;
    
```

```

procedure Register;

implementation

uses Forms, Registry;

var
    FlxpTrayIcon: TlxpTrayIcon;

procedure Register;
begin
    RegisterComponents('lxbpuaa', [TlxpTrayIcon]);
end;
    
```



6.3

托盘
组件



```
{ TlxpTrayIcon }
```

```
constructor TlxpTrayIcon.Create(AOwner: TComponent);
```

```
begin
```

```
  inherited Create(AOwner);
```

```
  FlxpTrayIcon := Self;
```

```
  FTrayIcon := TIcon.Create;
```

```
  FInterval := 500;
```

```
  TimerHandle := 0;
```

```
  FNotificationWnd := Classes.AllocateHWnd(NotificationWndProc);
```

```
{ 注意看下面的代码。没有使用 Application.MainForm 而使用 TForm(AOwner)。
```

```
  不过这也要求此组件必须放在应用程序的主窗体上 }
```

```
  if AOwner is TForm then
```

```
  begin
```

```
    OldWindowProc := TForm(AOwner).WindowProc;
```

```
    TForm(AOwner).WindowProc := NewWindowProc;
```

```
  end;
```

```
end;
```

```
procedure TlxpTrayIcon.NewWindowProc(var Message: TMessage);
```

```
begin
```

```
  if Assigned(OldWindowProc) then
```

```
    OldWindowProc(Message);
```

```
  with Message do
```

```
    if (Msg = WM_SYSCOMMAND) and
```

```
        (WParam = SC_MINIMIZE) then { 如果用鼠标点击窗口的最小化按钮，则隐藏窗口 }
```

```
        ShowWindow(Application.Handle, SW_HIDE);
```

```
end;
```

```
destructor TlxpTrayIcon.Destroy;
```

```
begin
```

```
  ShowTrayIcon(NIM_DELETE); { 删除托盘图标 }
```

```
  FreeAndNil(FTrayIcon);
```

```
  if FNotificationWnd <> 0 then
```

```
    Classes.DeallocateHWnd(FNotificationWnd); { 销毁窗口 }
```

```
  if TimerHandle <> 0 then
```

```
    KillTimer(0, TimerHandle); { "杀死" 定时器 }
```

```
  inherited Destroy;
```

```
end;
```

```

procedure TlxpTrayIcon.DoDbClick;
begin
    if Assigned(OnDbClick) then OnDbClick(Self);
end;

procedure TlxpTrayIcon.Loaded;
begin
    inherited;
    if not (csDesigning in ComponentState) then
    begin
        if FTrayIcon.Handle = 0 then
            FTrayIcon.Assign(Application.Icon);
        { 初始化 NotifyIconData }
        FillChar(NotifyIconData, SizeOf(NotifyIconData), 0);
        with NotifyIconData do
            begin
                cbSize := SizeOf(TNotifyIconData);
                Wnd := FNotificationWnd;
                uID := ICON_ID;
                uFlags := NIF_MESSAGE or NIF_ICON or NIF_TIP;
                uCallbackMessage := MI_ICONEVENT;
                hIcon := FTrayIcon.Handle;
                StrLCopy(szTip, PChar(Hint), SizeOf(szTip));
            end;
            ShowTrayIcon;
        end;
    end;

procedure TlxpTrayIcon.NotificationWndProc(var Message: TMessage);
var
    P: TPoint;
begin
    if Message.Msg = MI_ICONEVENT then
    begin
        case Message.LParam of
            WM_LBUTTONDOWNCLK:      { 如果双击托盘图标, 那么显示应用程序主窗口 }
            begin
                DoDbClick;
                RestoreApp;
            end;
            WM_RBUTTONDOWN:         { 如果右键点击托盘图标, 则显示弹出菜单 }
        end;
    end;

```



6.3

托盘
组件



```

begin
  if Assigned(FPopupMenu) then
  begin
    GetCursorPos(P);
    FPopupMenu.Popup(P.X, P.Y);
  end;
end;
end;
{ 对于别的消息, 则调用系统缺省方法处理之 }
end else Message.Result :=
  DefWindowProc(
    FNotificationWnd, Message.Msg, Message.WParam, Message.LParam);
end;

{ 显示动画图标 }
procedure SetAnimatedIcon(Wnd: HWND; Msg, idEvent: UINT; dwTime: DWORD);
  stdcall;
begin
  if Msg = WM_TIMER then with FlxpTrayIcon.NotifyIconData do
  begin
    if hIcon = 0 then hIcon := FlxpTrayIcon.FTrayIcon.Handle
    else hIcon := 0;
    Shell_NotifyIcon(NIM_MODIFY, @FlxpTrayIcon.NotifyIconData);
  end;
end;

{ 图标的增加、修改、删除都实现在 ShowTrayIcon 过程中 }
procedure TlxpTrayIcon.ShowTrayIcon(
  Mode: Cardinal = NIM_ADD; Animated: Boolean = False);
begin
  if csDesigning in ComponentState then Exit;
  if Mode = NIM_MODIFY then
  begin
    if Animated then
    begin
      if TimerHandle = 0 then
        TimerHandle := SetTimer(0, 0, FInterval, @SetAnimatedIcon)
      end else if TimerHandle <> 0 then
      begin
        KillTimer(0, TimerHandle);
        TimerHandle := 0;
      end;
    end;
  end;
end;

```

6.3 托盘组件

```

    then
    begin
        if B then
            Reg.WriteString(KeyName,Application.ExeName)
        else Reg.DeletetKey (KeyName);
        Reg.CloseKey;
    end;
finally
    FreeAndNil(Reg);
end;
end;

procedure TlxpTrayIcon.Notification(
    AComponent: TComponent; Operation: TOperation);
begin
    inherited Notification(AComponent, Operation);
    if Operation = opRemove then
    begin
        if AComponent = FPopupMenu then FPopupMenu := nil;
    end;
end;

end.

```

小结

本节开发了一个完善的托盘组件，该组件可以直接使用在正式开发中。

本节融汇了较多的开发技巧，如：构建窗口和窗口过程；运用 API 函数实现定时器；用嫁接法处理消息；注册表操作等。

6.4 自动下拉的 TComboBox

当你在 IE 的地址栏键入字符时，它自动将以已键入字符串作为开头的历史地址全部列出来。这样的功能是不是很体贴人心？

心动不如行动，其实我们也可以做一个这样的控件。让我们向自己的梦想进发吧。

我们直接从 TComboBox 派生这个控件。编写这个控件的关键是：

- (1) 定义一个字符串列表保存所有原始字符串，就好比 IE 保存的所有历史地址。这样可以在用户键入字符时，将所有满足条件的字符串加到 TComboBox.Items 中。
- (2) 实现键入时自动下拉，而不需要用户用鼠标选择下拉。

(3) 相同的字符串应该作为一项而不是多项显示。
这个组件比较简单，所以我们在下面直接列出源代码：

```
unit lxpAutoListComboBox;

interface

uses
  Windows, SysUtils, Classes, Controls, StdCtrls;
type
  TlxpAutoListComboBox = class(TComboBox)
  private
    FText: String;
    { FallStrings 用来保存所有的字符串。如果数量很多的话，你可以使用
      THashedStringList 而不是 TStrings，这样可以加快速度 }
    FallStrings: TStrings;
    procedure SetStrings(const Value: TStrings);
  protected
    { 覆盖事件驱动方法 DoEnter。当控件获得焦点时，作一些初始化工作 }
    procedure DoEnter; override;
    { 覆盖事件驱动方法 KeyPress。附加初始化，并自动下拉列表 }
    procedure KeyPress(var Key: Char); override;
    { 当用户键入字符导致 Text 变化时，改变 Items 包含的字符串 }
    procedure Change; override;
  public
    property AllStrings: TStrings read FallStrings write SetStrings;
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  end;

procedure Register;

implementation

uses Messages;

procedure Register;
begin
  RegisterComponents('lxpbuaa', [TlxpAutoListComboBox]);
end;
```



```

{ TlxpAutoListComboBox }

constructor TlxpAutoListComboBox.Create(AOwner: TComponent);
begin
    inherited;
    {TComboBox 的 AutoComplete 属性的实现有些问题, 如果为 True, 将使这个组件乱七八糟, 所以 False 之}
    AutoComplete := False;
    FallStrings := TStringList.Create;
end;

destructor TlxpAutoListComboBox.Destroy;
begin
    FreeAndNil(FallStrings);
    inherited;
end;

procedure TlxpAutoListComboBox.SetStrings(const Value: TStrings);
begin
    if Assigned(FallStrings) then
        FallStrings.Assign(Value)
    else
        FallStrings := Value;
end;

{ 局部过程 PackStrings 用来压缩一个字符串列表。这里的压缩是指去掉重复的项。注意这个过程没有声明只有实现, 因此是局部的。这样的过程和函数只有在实现位置之后才能调用。 }
procedure PackStrings(Str: TStrings);
var
    I: Integer;
    S: String;
begin
    if Str.Count = 0 then Exit;
    { 在更新 TStrings 的大量项时, 总是将更新工作包在 BeginUpdate 和 EndUpdate 之间, 以避免 TStrings 的一些事件发生, 从而提高处理速度 }
    Str.BeginUpdate;
    { 因为 TStrings 是一个抽象类, 创建一个 TStrings 实例时, 总是使用 TStringList 及其子类, 所以我们可以放心地将 Str 强制转化为 TStringList。转化的目的是使用 TStringList 的排序功能 }
    with TStringList(Str) do

```

6.4 自动下拉的 TComboBox



```

begin
  inherited Change;
  if Text = '' then
  begin
    Items.Assign(FAllStrings);
    Exit;
  end;
  if Items.IndexOf(Text) <> -1 then Exit;

  FText := Text;
  ISelStart := SelStart;
  Items.Clear;      { 不能放在上两句之前 }
  MaxWidth := 0;
  IndexMaxWidth := -1;
  for I := 0 to FAllStrings.Count-1 do
  if Pos(FText, FAllStrings[I]) > 0 then
  begin
    Items.Add(FAllStrings[I]);
    IWidth := Length(FAllStrings[I]);
    if IWidth > MaxWidth then
    begin
      MaxWidth := IWidth;
      Inc(IndexMaxWidth); { 将列表中字符数最多的项的位置保存到 IndexMaxWidth }
    end;
  end;
  for I := 1 to 8-Items.Count do
    Items.Add('');
    { 改变列表项显示的长度，以便可以完全显示最长的字符串 }
    Perform(CB_SETDROPPEDWIDTH,
      Canvas.TextWidth(Items[IndexMaxWidth])+10,0);
    if not DroppedDown then
      DroppedDown := True;
    Text := FText;
    SelStart := ISelStart;
  end;
end.

```

这个控件使用时如图 6-2 所示。



图 6-2 自动下拉列表框

小结

开发 TlxpAutoListComboBox 控件的要点是：

- (1) 在焦点进入控件 (DoEnter) 时，下拉列表，可以避免鼠标隐形问题。
- (2) 在控件文本变化 (Change) 时，改变列表的字符串项。
- (3) 如何去掉字符串列表的相同项 (PackStrings 过程)。

6.5 开发数据敏感控件

数据敏感控件一般的开发方式是给一个控件加上数据敏感功能。如 VCL 自带的 TDBEdit、TDBImage 等，都是在对应的不具备数据敏感功能的控件上扩展起来的。

控件的数据敏感能力分为几种：只读 (浏览) 型、读写型以及多记录型。只读型的一般用处不大，因此，绝大多数是读写型的；多记录型在 VCL 中比较典型的是 TDBGrid、TDBCtrlGrid 以及 TDBChart，其实现比较复杂。

很多人也许会认为，数据敏感控件作为 VCL 中很重要的一类组件，应该有一些通用的方法来存取数据信息，如绑定到哪个字段。而不是像现在这样，每编写一类数据敏感控件，都需要重新发布 Field 属性，并自己处理有关它的一切问题。

VCL 似乎应该在 TControl 级就实现 Field 属性，或者采用相对隐蔽一些的方法，如像 .NET Framework 那样动态绑定 (这样做，功能还要强大很多，比如理论上可以绑定到任何属性)。但由于历史的原因，这样做已经有了相当难度。因为 VCL 不是一开始就具备的数据敏感能力，而到后来需要控件的数据敏感能力时，这个架构已经确定了，就不好让它伤筋动骨。

本章希望通过开发一个读写型的日期敏感控件，让大家认识数据敏感控件：

- (1) 数据敏感原理。
- (2) 开发的一般步骤。

6.5.1 数据敏感原理

控件的数据敏感功能是通过使用辅助类 TFieldDataLink 来实现的。TFieldDataLink 定义在 DBCtrls 单元。首先，我们要弄清楚它的一些主要成员：

属性：

property FieldName: String;	对应的字段名。
property Field: TField;	对应的字段。
property DataSource: TDataSource;	对应的数据源。
property ReadOnly: Boolean;	读写状态（用以实现读写型的数据敏感控件）。
property CanModify: Boolean;	对应字段的数据是否可以修改。如果 ReadOnly=True 或者数据集指定该字段不能修改则 CanModify=False。
property Editing: Boolean;	对应字段是否在编辑状态。可以调用方法 Edit 设置它=True。
property Control: TComponent;	即要实现数据敏感功能的 TWinControl。

方法：

procedure Modified;	当控件的数据变化时，应调用此方法跟踪此变化。
procedure UpdateRecord;	用来生成 OnUpdateData 事件。数据编辑完成并需要提交数据时应调用此方法。
function Edit: Boolean;	使 DataSource 进入编辑状态。

事件：

property OnDataChange: TNotifyEvent;	属性 DataSource 连接的数据集数据变化时触发，在这里更新控件显示的数据。
property OnUpdateData: TNotifyEvent;	控件的数据被修改后需要提交到数据集时触发，在这里将修改后的数据按照一定规则整理后写到对应字段里。

综合起来，数据敏感的过程如下：

- (1) 设定 TFieldDataLink 对象的 Control、DataSource 和 FieldName 属性。
- (2) 当 OnDataChange 事件发生（如记录滚动）时，从 Field 取得当前数据，并显示在控件中。
- (3) 当控件显示的数据被修改时（一般是控件的 OnChange 事件），调用 Modified 方法跟踪。
- (4) 当控件的数据修改完毕需要提交时（一般是控件的 OnExit 事件或者 DoExit 消息驱动方法），调用 UpdateRecord 生成 OnUpdateData 事件。
- (5) 在 OnUpdateData 中更新 Field 中的数据。如果需要实时更新数据（如每按一个键时都更新），还可以在第 3 步直接更新 Field 中的数据。

6.5.2 开发日期敏感控件

由于 VCL 提供 TDateTimePicker 控件，所以可以从它派生一个类，然后使用辅助类 TFieldDataLink

来实现这个数据敏感控件。

根据上小节总结的开发过程，我们可以定义如下的类：

```

type
  TlxpDBDatePicker = class(TDateTimePicker)
  private
    FDataLink: TFieldDataLink;           {*}
    procedure DataChange(Sender: TObject);   {*}
    procedure UpdateData(Sender: TObject);   {*}
    procedure CMGetDataLink(var Message: TMessage); message
      CM_GETDATALINK;
    function GetDataField: String;
    procedure SetDataField(const Value: String);
    function GetDataSource: TDataSource;
    procedure SetDataSource(Value: TDataSource);
    function GetReadOnly: Boolean;
    procedure SetReadOnly(Value: Boolean);
    function GetField: TField;
  protected
    procedure Change; override;           {*}
    procedure DoExit; override;           {*}
    procedure KeyPress(var Key: Char); override;
    { 根据 ReadOnly 属性控制用户输入 }
    procedure Notification(
      AComponent: TComponent; Operation: TOperation); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    property Field: TField read GetField;
    { *, 公开一个运行时属性，方便存取字段 }
  published
    { *}
    property DataField: String read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write
      SetDataSource;
    property ReadOnly: Boolean read GetReadOnly write SetReadOnly default
      False; { 注意这里 default 的含义，它并不是用来指定属性默认值。请参看 3.3.3 }
  end;

```

上面代码中，*部分是重点，其他都是套路。框架已经定义完毕，在编辑区点击右键，选择“Complete class at cursor”，接着完成实现：





```

{ TlxpDBDatePicker }
constructor TlxpDBDatePicker.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    inherited ShowCheckbox:= True; { 显示勾选框, 这样可以实现去掉日期数据的功能 }
    Date := SysUtils.Date;          { 因为属性 Date 和函数 Date 同名, 所以在函数前面
                                     加上单元名字, 相当于"命名空间" }
    Width := 100;                   { 设置控件长度默认为 100 个像素 }
    FDataLink := TFieldDataLink.Create;
    FDataLink.Control := Self;       { 设置敏感控件 }
    FDataLink.OnDataChange := DataChange; { 设置 OnDataChange 事件处理过程 }
    FDataLink.OnUpdateData := UpdateData; { 设置 OnUpdateData 事件处理过程 }
end;

destructor TlxpDBDatePicker.Destroy;
begin
    FreeAndNil(FDataLink);
    inherited Destroy;
end;

procedure TlxpDBDatePicker.DataChange(Sender: TObject);
begin
    Checked:=Assigned(FDataLink.Field)and(FDataLink.Field.AsString<>'');
    { 当没有数据时, 勾选框显示为空 }
    try
        if Checked then { 如果有数据, 则显示它 }
            Date := StrToDateTime(FDataLink.Field.AsString);
    finally
    end;
end;

procedure TlxpDBDatePicker.Change;
begin
    if ReadOnly then { 在只读状态下, 不允许改变勾选状态 }
        begin
            if Checked <> (FDataLink.Field.AsString <> '') then
                Checked := not Checked;
            Exit;
        end;
end;

with FDataLink do { 跟踪数据变化 }

```

```

begin
    if not Editing then Edit;      { 最好在 OnKeyDown 和 OnKeyPress 事件中进入编辑状态。这里是为了简化}

    Modified;
end;
inherited;
end;

procedure TlxpDBDatePicker.DoExit;
begin
    try
        FDataLink.UpdateRecord;    { 激发 OnUpdateData 事件}
    except
        SetFocus;
        raise;
    end;
    inherited;
end;

procedure TlxpDBDatePicker.UpdateData(Sender: TObject);
var
    S: String;
begin
    if Checked then                { 如果在勾选状态，则向字段写入日期数据；否则写'' }
    begin
        try
            DateTimeToString(S, 'YYYY-MM-DD', Date);
            FDataLink.Field.AsString := S;
        except
            MessageBox(0, '不是日期型字段。', '提示', MB_ICONINFORMATION+MB_OK);
        end;
    end else FDataLink.Field.AsString := '';
end;

procedure TlxpDBDatePicker.KeyPress(var Key: Char);
begin
    if Key = #13 then
        keybd_event(VK_TAB, 0, 0, 0){ 如果用户按下回车键，则转化为跳格键}
    else if ReadOnly and
        (Ord(Key) in [48..57, 8, 45, 3, 22, 24]) then
        { 如果在只读状态，或者用户按下键不代表：数字、跳格、Esc 和 Ctrl+C/V/X，则不允许

```



```
    输入}
    Key := #0
  else
    inherited;
  end;

{ 以下都比较简单，但却是编写数据敏感控件必不可少的代码}
function TlxpDBDatePicker.GetField: TField;
begin
  Result := FDataLink.Field;
end;

function TlxpDBDatePicker.GetDataField: String;
begin
  Result := FDataLink.FieldName;
end;

procedure TlxpDBDatePicker.SetDataField(const Value: String);
begin
  FDataLink.FieldName := Value;
end;

function TlxpDBDatePicker.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TlxpDBDatePicker.SetDataSource(Value: TDataSource);
begin
  if not (FDataLink.DataSourceFixed and (csLoading in ComponentState))
  then
    FDataLink.DataSource := Value;
    if Value <> nil then Value.FreeNotification(Self);
    { 辅助对象应该向主对象注册销毁通知}
  end;

function TlxpDBDatePicker.GetReadOnly: Boolean;
begin
  Result := FDataLink.ReadOnly;
end;
```

```

procedure TlxpDBDatePicker.SetReadOnly(Value: Boolean);
begin
    FDataLink.ReadOnly := Value;
end;

procedure TlxpDBDatePicker.CMGetDataLink(var Message: TMessage);
begin
    Message.Result := Integer(FDataLink);
end;

procedure TlxpDBDatePicker.Notification(AComponent: TComponent;
    Operation: TOperation);
begin
    inherited Notification(AComponent, Operation);
    if (Operation = opRemove) and (FDataLink <> nil) and
        (AComponent = DataSource) then DataSource := nil;
end;

end.

```

到目前为止，一个数据敏感控件已经开发出来了。不过这个控件还有两个问题没有解决：

(1) 由于 TDateTimePicker 除了键入外, 还可以通过下拉框来改变日期。因此, 在这两个方面都要解决 ReadOnly 问题。很明显, 在只读状态下, 是不能让用户通过选择来改变数据的。

(2) 当数据源中的日期数据为空时，我们的控件仍然会显示某个日期，只不过勾选框未被选中，这无疑也是美中不足。

问题(1)的解决办法是,在用户选择一个新的日期后,如果是只读状态,则应该恢复选择前的日期。可以在 OnCloseUp 事件中处理。

问题(2)更简单，在所有控件日期改变的地方设置 `TDateTimePicker.Format` 属性就可以了。也许有的朋友会认为在 `ReadOnly` 属性改变时，直接修改 `Enabled` 属性就可以达到目的了，而且简单实用！这的确不失为一个办法，不过可能会激起用户的恼怒，比如他并不想真的修改数据，只不过想看看那个下拉框是什么样子的時候。况且 VCL 已有的敏感控件也都没有采取这样的做法，所以我们还是不要如此粗鲁吧！

我们需要在 `private` 域增加几个变量和方法：

```
type
    TlxpDBDatePicker = class(TDateTimePicker)
    private
        .....
end type
```

6.5

开发数据敏感控件


```

PreDate: TDate;           { 保存选择前的日期}
PreOnDropDown,PreOnCloseUp:TNotifyEvent;{ 保存 OnDropDown 和 OnCloseUp
                                   事件处理过程指针}

procedure NewDropDown(Sender: TObject); { 新的 OnDropDown 处理过程}
procedure NewCloseUp(Sender: TObject); { 新的 OnCloseUp 处理过程}
procedure SetFormat;           { 改变 Format 属性}
.....

.....
implementation
.....

procedure TlxpDBDatePicker.NewDropDown(Sender: TObject);
begin
  if Assigned(PreOnDropDown) then PreOnDropDown(Sender);
  PreDate := Date;           { 保存选择前的日期}
end;

procedure TlxpDBDatePicker.NewCloseUp(Sender: TObject);
begin
  if Assigned(PreOnCloseUp) then PreOnCloseUp(Sender);
  if ReadOnly then           { 在只读状态, 恢复选择前的日期}
  begin
    if PreDate <> Date then Date := PreDate;
    Checked := FDataLink.Field.AsString <> '';
  end;
  SetFormat;
end;

{ 除了NewCloseUp, 还应该 DataChange 的"finally...end"中和 Change 的最后调用
  SetFormat }
procedure TlxpDBDatePicker.SetFormat;
begin
  if Checked then Format := '' { 如果日期数据存在, 则显示; 否则显示空}
  else Format := 'gg';
end;

```

小结

本节讲述了开发数据敏感控件的全面知识。通过本节概念和实例的学习, 今后开发数据敏感控件就不会有什么大的疑难问题了。

6.6 开发聚合组件

开发数据库软件时，常常需要在很多数据敏感控件上方或者左边放一个 TLabel，其 Caption 设置为 TField.DisplayLabel，用于标识敏感控件对应的字段。不过这种开发有个弊病，就是我们总要不停地调 TLabel 和数据敏感控件的位置、大小和间距什么的，而且不准确；再有，修改了永久字段的 DisplayLabel 以后，还需要更改每个 Label 的 Caption。这真是折磨人。我们多么希望有一个带 Label 的敏感控件，它可以自己调位置，还可以从 TField.DisplayLabel 自动取标识啊！这样就可以彻底解脱了！

在本节里，首先就将一个 Label 连到 TlxpDBDatePicker 上，完成一个新的组件 TlxpLabelDBDatePicker，这种开发方法叫做聚合。接下来，还要完成自动取标识的功能。

6.6.1 开发 LabelDBDatePicker

TlxpDBDatePicker 可以如下设计：

```

type
  TlxpLabelDBDatePicker = class(TlxpDBDatePicker)
  private
    FLabel: TLabel;           { 辅助对象 }
    FLabelSpacing: Integer;   { 辅助对象和主对象的间距 }
    procedure SetLabelSpacing(const Value: Integer);
  protected
    { 以下几个详细方法用于保持辅助对象和主对象的互动 }
    procedure CMVisiblechanged(var Message: TMessage);
    message CM_VISIBLECHANGED;
    procedure CMEnabledchanged(var Message: TMessage);
    message CM_ENABLEDCHANGED;
    procedure CMBidimodechanged(var Message: TMessage);
    message CM_BIDIMODECHANGED;
    procedure CMFontchanged(var Message: TMessage);
    message CM_FONTCHANGED;
    procedure SetParent(AParent: TWinControl); override;
    { 因为主对象被设定 Parent 时，才能设定辅助对象的 Parent }
    procedure Notification(AComponent: TComponent; Operation: TOperation);
    override;
  public
    procedure SetBounds(ALeft: Integer; ATop: Integer; AWidth: Integer;
      AHeight: Integer); override;
    { 覆盖主类的 SetBounds，当主对象位置和大小发生改变时，该方法会被自动调用。因此，
      可以在这时候对辅助对象的位置大小作相应调整 }
  end;

```



```

constructor Create(AOwner: TComponent); override;
published
  property DisplayLabel: TLabel read FLabel;
    { 显示标识 (对应 TField.DisplayLabel) 属性 }
  property LabelSpacing: Integer read FLabelSpacing write
    SetLabelSpacing;      { 辅助对象和主对象的间距属性 }
end;

```

好,框架已经定义完毕,下面我们就来逐个实现这些方法。在编辑区点击右键,选择“Complete class at cursor”。开始编码:

```

constructor TlxpLabelDBDatePicker.Create(AOwner: TComponent);
begin
  inherited;
  FLabel := TLabel.Create(Self);
  { FLabel 的 Owner 为主对象,因此,主对象被销毁时,FLabel 也会被销毁;我们不需要覆盖
    主类的 Destroy 来显式销毁 FLabel }
  FLabel.Caption := 'DisplayLabel';
  FLabel.FreeNotification(Self);
  FLabelSpacing := 3;      { 指定默认间隔为 3 个像素 }
end;

procedure TlxpLabelDBDatePicker.SetLabelSpacing(const Value: Integer);
begin
  if Value <> FLabelSpacing then
  begin
    FLabelSpacing := Value;
    { 我们将 FLabel 放在主控件左边,所以位置间隔变换很简单。你还可以再发布一个属性,
      让用户指定 FLabel 可以放在主控件的上下左右。 }
    FLabel.Left := Left - (FLabelSpacing + FLabel.Width);
  end;
end;

procedure TlxpLabelDBDatePicker.SetParent(AParent: TWinControl);
begin
  inherited;
  { 不要将这句移到其他地方。SetParent 是 TControl 将要设置 Parent 时调用的一个虚
    拟方法,在这里可以确保主控件的每次 Parent 变化都可以捕捉到;而且在其他地方,
    主控件的 Parent 可能还不存在呢 }
  FLabel.Parent := AParent;
end;

```

{ 以下几个方法用于保持辅助对象和主对象的互动}

```
procedure TlxpLabelDBDatePicker.CMBidimodechanged(
  var Message: TMessage);
begin
  inherited;
  FLabel.BiDiMode := BiDiMode;
end;
```

```
procedure TlxpLabelDBDatePicker.CMEnabledchanged(
  var Message: TMessage);
begin
  inherited;
  FLabel.Enabled := Enabled;
end;
```

```
procedure TlxpLabelDBDatePicker.CMFontchanged(var Message: TMessage);
begin
  inherited;
  { Font 是一个对象，这里采用复制 (Assign) 而不是指针引用 (FLabel.Font := Font) ,
    是为了避免可能的引用混乱。 }
  FLabel.Font.Assign(Font);
end;
```

```
procedure TlxpLabelDBDatePicker.CMVisiblechanged(
  var Message: TMessage);
begin
  inherited;
  FLabel.Visible := Visible;
end;
```

```
procedure TlxpLabelDBDatePicker.SetBounds(
  ALeft, ATop, AWidth, AHeight: Integer);
begin
  inherited;
  { 这里必须首先判断 FLabel 是否存在，因为主对象第一次调用 SetBounds 时，FLabel
    还没有创建。 }
  if FLabel <> nil then with FLabel do
  begin
    Top := ATop + (AHeight - Height) div 2;
    Left := ALeft - (FLabelSpacing + Width);
```





```

    end;
end;

procedure TlxpLabelDBDatePicker.Notification(
    AComponent: TComponent; Operation: TOperation);
begin
    inherited;
    if (Operation = opRemove) and (AComponent = FLabel) then
        FLabel := nil;
end;

```

这样就完成了聚合控件 TlxpLabelDBDatePicker。使用它，可以减省大量的无意义劳动。

6.6.2 加强 LabelDBDatePicker

下面来实现 TlxpLabelDBDatePicker 的自动取标识功能。

在 6.3.7 中已经介绍了 TComponent 的虚拟方法：

```
procedure Loaded; virtual;
```

现在又是用到它的时候了。我们说过：“当所有组件被创建，并从 dfm 文件读出数据初始化这些组件实例后，Loaded 方法被自动调用”。通常情况下，我们在数据集中使用字段时，要指定每个字段的 DisplayLabel，且数据集放在数据模块（DataModule）中。而数据模块常常是在程序启动不久、先于数据显示编辑窗口被创建。换句话说，数据敏感控件被创建前，它对应的字段已经被创建了。所以我们在敏感控件的 Loaded 中，就可以取得字段的 DisplayLabel 被写到 DisplayLabel.Caption。

这就是一个自动取标识的过程。这样做的好处是，不需要给每个自带 Label 的数据敏感控件人工设置 DisplayLabel.Caption，即使修改字段的 DisplayLabel 后也不用管它；因为工程无论是在设计时还是在运行时，这些数据都会被自动取得。

在 TlxpLabelDBDatePicker 的 protected 域覆盖 Loaded：

```

protected
    procedure Loaded; override;

```

然后实现：

```

procedure TlxpLabelDBDatePicker.Loaded;
begin
    inherited;
    if Field <> nil then
        FLabel.Caption := Field.DisplayLabel;    { 自动取得标识 }
end;

```

根据本小节开始的分析，我们知道必须要保证字段先于数据敏感控件被创建。而现在呢，数据集和敏感控件数据都在一个 Form 上，所以这个条件得不到保证。因此，使用这个控件时，要特别注意这一点。

其实还没完。完美主义者总是有很多话要说。我们有没有想过：如果在程序运行时修改了字段的 `DisplayLabel` 会怎么样？这时候敏感控件的标识就不会跟着变了，因为 `Loaded` 只在最初执行一次！难道这时候要弹出一个对话框：“喂，亲爱的用户，你需要重新启动程序”吗？然后用户说：“你去死吧，我不再用你的软件了！”所以这样搞的话，是不得人心的。

值得注意的是，窗口的激活事件是 `OnActivate` 而不是 `OnShow`。一个窗口只要没有被 `Hide`，它就永远在 `Show` 状态（即使当前顶层窗口不是它）。只有当焦点从另一个窗口转移到某窗口时，这个窗口的 `OnActivate` 才会被激发。所以在这里，我们应该使用 `OnActivate` 而不是 `OnShow`。

开发聚合组件

开发聚合组件

```

procedure TlxpLabelDBDatePicker.Loaded;
begin
    inherited;
    { 以下注释掉的代码可以不要了, 因为我们在 GetDisplayLabel 中实现相同的功能 }
    { if Field <> nil then
        FLabel.Caption := Field.DisplayLabel; }

```

```

if Owner.InheritsFrom(TForm) then with TForm(Owner) do
begin
  { 首先保存 Form 原来的 OnActivate 句柄, 然后赋予新方法 }
  OldOnActivate := OnActivate;
  OnActivate := GetDisplayLabel;
end;
end;

procedure TlxpLabelDBDatePicker.GetDisplayLabel(Sender: TObject);
begin
  { 如果原来的 OnActivate 处理过程存在, 则首先执行它 }
  if Assigned(OldOnActivate) then OldOnActivate(Sender);
  if Field <> nil then
    FLabel.Caption := Field.DisplayLabel;
end;

```

好, 写完了, 编译, 打开一个例子运行。结果与我们预想的一样, 即使数据集和敏感控件在一个窗体上也能让敏感控件正确取得标识, 原来的限制也被去掉了!

善于从全局、把多个事物联系起来考虑问题的人看到这里, 可能有一个疑问。如果 Form 上放了多个这样的敏感控件, 那么上面的代码会不会造成重复调用, 使 Form.OnActivate 执行多次? 乍一看的确有这种嫌疑, 但仔细分析后发现是没有这种可能的。现在假设放了两个敏感控件, 分别为 C1、C2, 对应字段为 F1、F2。另外也假设 Form.OnActivate 是存在的, 用 OldOnActivate 表示。

则 C1.Loaded 执行后 Form.OnActivate 变为:

```

begin
  OldOnActivate(Sender);
  if F1 <> nil then {.....}
end;

```

紧接着, C2.Loaded 执行后 Form.OnActivate 变为:

```

begin
  OldOnActivate(Sender);
  if F1 <> nil then {.....}
  if F2 <> nil then {.....}
end;

```

因此, 最终 Form.OnActivate 的代码是:

```
begin
  { 首先执行 OnActivate }
  { 然后逐个给所有敏感控件取标识 }
end;
```

通过上面的分析我们知道，同一个窗体上的所有敏感控件取标识的代码最终放到了一起，且是一次执行完成的，是不是很爽？！

老实说，我们的敏感控件做到现在这个样子，已经很不错了。但是还有一个不足，不知道大家注意到没有：如果运行时我在同一个窗体中修改了字段 DisplayLabel，则这个窗体上的敏感控件标识不会马上更新，必须要重新 Activate 以后才行！在不修改组件源代码的情况下，可以有两种方法马上更新标识：

(1) 修改字段 DisplayLabel 的代码后，直接调用 Form.OnActivate。这个调用是直接执行一段代码，不会真的让窗体发生 Activate 事件。这样很简单，也很有效率。

(2) 修改字段 DisplayLabel 后利用 RTTI 写一段程序马上更新窗体上所有敏感控件的标识。大家看我写的一个过程，它可以刷新一个父控件上所有子控件（包括子、子的子、子的孙等）中为数据敏感控件者的 DisplayLabel：

```
uses TypInfo, DB;

{ Parent 参数就是上面所说的父控件。比如你要刷新 Form1 上所有敏感控件标识，那么传入
  Form1；如果是 Panel1，传入 Panel1。 }
procedure RefreshDBControlLabel(Parent: TWinControl);
var
  I: Integer;
  tFoundCtr: TControl;
  PropInfo: PPropInfo;
  FieldName: String;
  FoundField: TField;
  FoundDataSource: TDataSource;
const
  { 定义三个常数，表示数据敏感控件必需的三个属性的名字 }
  PropName1 = 'DisplayLabel';
  PropName2 = 'DataField';
  PropName3 = 'DataSource';
begin
  for I := 0 to Parent.ControlCount-1 do
  begin
    tFoundCtr := Parent.Controls[I];
    PropInfo := GetPropInfo(tFoundCtr, PropName1);
```




```

if PropInfo <> nil then      { 如果有'DisplayLabel'属性 }
begin
  PropInfo := GetPropInfo(tFoundCtr, PropName2);
  if PropInfo <> nil then    { 如果有'DataField'属性 }
  begin
    FieldName := GetStrProp(tFoundCtr, PropName2);
    if FieldName <> '' then { 如果'DataField'属性的值不为空 }
    begin
      PropInfo := GetPropInfo(tFoundCtr, PropName3);
      if PropInfo <> nil then { 如果有'DataSource'属性 }
      begin
        FoundDataSource := TDataSource(GetObjectProp
          (tFoundCtr, PropName3));
        if FoundDataSource <> nil then { 如果'DataSource'属性对象存在 }
        begin
          FoundField := FoundDataSource.DataSet.FindField(FieldName);
          { 如果数据集中存在该字段, 注意不要用FieldByName 而使用FindField。
            FieldByName 在找不到指定名字字段的情况下会触发异常, 而FindField
            不会 }
          if FoundField <> nil then
            { 刷新敏感控件标识 }
            TLabel(GetObjectProp(tFoundCtr, PropName1)).Caption :=
              FoundField.DisplayLabel;
        end;
      end;
    end;
  end;
end;
end;
end;
{ 最后这几句很重要, "子控件(包括子、子的子、子的孙等)" 这样一个功能就是由这句完
成的。它做了一个递归: 如果当前处理的控件是TWincontrol (这种控件可以作为其他
控件的父控件), 调用原过程再处理它所有的子控件。 }
if tFoundCtr is TWincontrol then
  RefreshDBControlLabel(TWincontrol(tFoundCtr));
end;
end;

```

喜欢刨根问底的人会说:“你这个自动取标识的功能还不彻底”。假设我使用一个内存编辑器直接修改你那个程序的内存, 改了字段的 DisplayLabel, 嘿嘿, 这下你根本就不知道我已经改了它, 可怎么更新标识呢! 呵呵, 的确是这样! 不过应该很少人会这么干吧。当然解决办法还是有的(以下内容权当吹牛了, 不必当真)。在“VCL 消息大全”中介绍了一个消息 WM_PAINT。WM_PAINT 的接收

和处理是持续不断的，两次之间差不多也就相差 n 分之一秒，如果将更新标识的代码写在这个消息方法里面，就可以解决这个问题，不管你用什么改、怎么改！

重新声明这个消息方法，然后实现：

```
protected
  procedure WMPaint (var Message: TMessage); message WM_PAINT;

  procedure TlxpLabelDBDatePicker.WMPaint(var Message: TWMPaint);
  begin
    inherited;
    if Field <> nil then
      FLabel.Caption := Field.DisplayLabel;
    end;
```

这样做的话，连修改 Form.OnActive 那一段代码都可以去掉了，反正 WMPaint 要不停地、永无休止地执行嘛！

不过请你不要真的这么做，基本上是没有必要。因为 WMPaint 频繁地执行，但我们仅仅偶尔更改字段标识，所以绝大多数时候 WMPaint 是在做无用功——不但无用，而且消耗资源。

小结

本节融汇了开发聚合组件时需要掌握的众多要点，重点要理解：

- (1) Parent 和 Owner 的区别以及相关的 Loaded、SetParent 等方法。
- (2) 如何让辅助对象和主对象互动，包括外观变化、大小位置修正等以及 SetBounds 等方法。
- (3) 使用 RTTI 操作没有共同父类的数据敏感控件群。

6.7 开发图形图像控件

要在控件上绘出图形或者图像，必须具有画布对象。

VCL 中内置画布对象的基本类主要是：TGraphicControl 和 TCustomControl。TGraphicControl 用于无窗口图形控件，TCustomControl 用于有窗口图形控件。所以，如果我们需要开发图形图像控件，一般从这两个类或其子类派生。

如果要在普通窗口控件上加入图形图像能力，可以给它增加一块画布（一般从 TControlCanvas 创建），然后处理 WM_PAINT 消息，在该消息方法中绘制图形图像。

如果要提升现有图形图像控件的图形图像处理能力，可以覆盖它的虚拟方法：

```
procedure Paint; virtual;
```

在该方法内部操纵 Canvas，进行自定义图形图像绘制。Paint 其实就是响应 WM_PAINT 消息，只



不过是父类在 WM_PAINT 消息方法中调用了它。

我们从 TGraphicControl 派生一个图形控件，用于显示艺术字体。下面是整个类的声明部分：

```

type
  { 支持三种艺术效果：3D、阴影、外框 }
  TArtWordStyles = set of (aws3D, awsShadow, awsOutlined);
  TlxpArtWordLabel = class(TGraphicControl)
  private
    FArtWordStyles: TArtWordStyles;
    { Caption 的对齐方式 }
    FAlignment: TAlignment;
    { TControl 有保护的 Caption 属性，我们本可以直接发布，但由于设置 Caption 时
      需要作些处理，故重新声明 }
    FCaption: TCaption;
    FAutoSize: Boolean;
    { 分别使用什么颜色显示 3D、阴影、外框效果 }
    FColor3D, FColorShadow, FColorOutline: TColor;
    { 阴影和本体的距离 }
    FShadowLength: Integer;
    procedure SetArtWordStyles(Value: TArtWordStyles);
    procedure SetCaption(Value: TCaption);
    function GetFont: TFont;
    procedure SetFont(Value: TFont);
    function GetColor(index: Integer): TColor;
    procedure SetColor(index: Integer; Value: TColor);
    procedure SetShadowLength(Value: Integer);
    procedure SetAlignment(Value: TAlignment);
  protected
    { 覆盖父类保护的 Paint 方法，绘出我们需要的效果 }
    procedure Paint; override;
    { SetAutosizes 是 AutoSize 属性的写方法，由于在父类已经存在，所以我们覆盖它(但
      实现时不一定需要继承父类的处理) }
    procedure SetAutoSize(Value: Boolean); override;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property Caption: TCaption read fCaption write SetCaption;
    { Font 属性没有存取字段，而代之以方法；存取方法和 Canvas.Font 交互 }
    property Font: TFont read GetFont write SetFont;
    property ArtWordStyles: TArtWordStyles read FArtWordStyles write

```

```

property Align;
property ParentFont;
property ParentShowHint;
property ShowHint;
property Visible;
property OnClick;
property OnDbClick;
property OnDragDrop;
property OnDragOver;
property OnEndDrag;
property OnMouseDown;
property OnMouseMove;
property OnMouseUp;
property OnStartDrag;
end;

```

6.7 开发图形图像控件

```
implementation

{ 只在实现部分用到的单元不应该写到声明部分的uses 语句中 }

uses Math;

constructor TlxpArtWordLabel.Create(AOwner: TComponent);
begin
    inherited;
    { 指定一些属性的默认值 }
```

```
FArtWordStyles := [aws3D, awsShadow];
FCaption := 'TlxpArtWordLabel';
fColor3D := clWhite;
FColorShadow := clGray;
FColorOutline := clWhite;
FShadowLength := 3;
end;

procedure TlxpArtWordLabel.SetCaption(Value: TCaption);
begin
  { 设置属性前一般应该检查设置值和已有值是否相同, 如果相同就不需要设置了。}
  if FCaption <> Value then
  begin
    FCaption := Value;
    { 调用 Repaint 执行重画。Repaint 内部调用 Paint。以下所有 Set 方法中都调用了
      Repaint, 因为设置这些属性后的显示效果需要重新绘出}
    Repaint;
  end;
end;

function TlxpArtWordLabel.GetFont: TFont;
begin
  Result := Canvas.Font;
end;

procedure TlxpArtWordLabel.SetFont(Value: TFont);
begin
  if Canvas.Font <> Value then
  begin
    Canvas.Font := Value;
    Repaint;
  end;
end;

procedure TlxpArtWordLabel.SetArtWordStyles(Value: TArtWordStyles);
begin
  if FArtWordStyles <> Value then
  begin
    FArtWordStyles := Value;
    Repaint;
  end;
end;
```

```

end;

{ 大家可以仔细看看带 index 指令的属性是如何存取的 }
function TlxpArtWordLabel.GetColor(index: Integer): TColor;
begin
    case index of
        0: Result := FColor3D;
        1: Result := FColorShadow;
        2: Result := FColorOutline;
    end;
end;

procedure TlxpArtWordLabel.SetColor(index: Integer; Value: TColor);
begin
    if GetColor(index) <> Value then
    begin
        case index of
            0: FColor3D := Value;
            1: FColorShadow := Value;
            2: FColorShadow := Value;
        end;
        Repaint;
    end;
end;

procedure TlxpArtWordLabel.SetShadowLength(Value: Integer);
begin
    if FShadowLength <> Value then
    begin
        FShadowLength := Value;
        Repaint;
    end;
end;

procedure TlxpArtWordLabel.SetAutosize(Value: Boolean);
begin
    if FAutoSize <> Value then
    begin
        FAutoSize := Value;
        { TextWidth 表示在画布上以当前字体设置输出字符串需要的宽度(像素)。4 是预留的边
          距 }
    end;
end;

```



```

    Width := Canvas.TextWidth(FCaption) + 4;
    { TextHeight 表示在画布上以当前字体设置输出字符串需要的高度(像素) }
    Height := Canvas.TextHeight(FCaption) + 4;
    Repaint;
end;
end;

procedure TlxpArtWordLabel.SetAlignment(Value: TAlignment);
begin
    if FAlignment <> Value then
    begin
        FAlignment := Value;
        Repaint;
    end;
end;

{ Paint 是本控件的核心部分 }
procedure TlxpArtWordLabel.Paint;
var
    Backup: TColor;
    X, Y: Integer;
begin
    inherited;

    { 输出的 Caption 在纵向始终位于中部 }
    Y := Max((Height - Canvas.TextHeight(FCaption)) div 2, 0);
    { 根据对齐方式算出 Caption 在横向的起始输出位置 }
    case FAlignment of
        taCenter:
            X := Max((Width - Canvas.TextWidth(FCaption)) div 2, 0);
        taLeftJustify:
            X := 2;
        taRightJustify:
            X := Width - Canvas.TextWidth(FCaption) - 2;
    end;

    { 设置 Style := bsClear 以避免闪烁 }
    Canvas.Brush.Style := bsClear;
    { 首先保存原来字体颜色, 因为下面实现各种效果时需要临时改变颜色, 最后我们需要恢复它,
      为下一次绘画所用 }
    Backup := Font.Color;

```

```

if aws3D in FArtWordStyles then with Canvas do
begin
  Font.Color := FColor3D;
  { 向右下移动一个像素单位, 绘出FCaption }
  TextOut(X + 1, Y + 1, FCaption);
  { 恢复Font.Color, 下面的流程需要使用。以下同 }
  Font.Color := Backup;
end;

if awsShadow in FArtWordStyles then with Canvas do
begin
  Font.Color := FColorShadow;
  { 向右下移动FShadowLength 各像素单位, 绘出FCaption }
  TextOut(X + FShadowLength, Y + FShadowLength, FCaption);
  Font.Color := Backup;
end;

if awsOutlined in FArtWordStyles then with Canvas do
begin
  Font.Color := FColorOutline;
  { 分别向四边外移一个像素单位, 绘出FCaption }
  TextOut(X + 1, Y + 1, FCaption);
  TextOut(X - 1, Y + 1, FCaption);
  TextOut(X + 1, Y - 1, FCaption);
  TextOut(X - 1, Y - 1, FCaption);
  Font.Color := Backup;
end;

{ 最后在起始位置绘出FCaption。由于它最后绘出, 所以会遮盖上面绘出的三种FCaption;
  最终绘出的FCaption 和未被遮盖的FCaption 部分共同作用形成最终效果。 }
Canvas.TextOut(X, Y, FCaption);
end;

```

测试时, 可以设置将字体设置大一些, 效果会更明显。

小结

图形图像控件是 VCL 的一个重要组成部分。本节通过开发一个简单艺术字控件来演示了如何开发图形图像控件。

当一些影响图形图像控件外观的属性发生变化时, 通常要调用以下一些方法重新绘制控件表面,





以响应属性变化：

(1) Invalidate。将控件表面标记为过时(或者说无效),需要重新绘制。但是需要调用 Update 才能实现绘制。

(2) Update。调用它们后会重新绘制控件表面的过时部分。

(3) Repaint、Refresh。这两个方法的作用是相同的。调用它们后会重新绘制整个控件表面。

以上几个方法的含义是针对 TWinControl 而言的,注意它们对于不同的类是有所不同的。关于这些方法在别的类(如 TControl)中的具体含义,请参考 Delphi 的在线帮助。

6.8 开发 QuickReport 组件

QuickReport 是 Borland 外购的组件包,Delphi 中并没有 QuickReport 组件的源代码。

在 Delphi 7 中,采用新的 Rave 组件来设计报表,缺省情况下 QuickReport 包没有安装。但是我们可以通过以下方法加上它：

选择菜单 Component|Install Packages|Add,然后再选择 Delphi 7\bin\dclqrt70.bpl,按“打开”按钮。

我们发现 QuickReport 组件包没有可以画斜线的控件,而本节的任务就是开发一个画斜线的 QuickReport 控件。

该从哪里着手呢?逐个观察已有组件,我们发现 TQRShape 可以画出横向和纵向直线。呵呵,画水平线、垂直线与斜线并没有本质区别,都是调用 TCanvas.MoveTo 将画笔移动到直线起点,再调用 TCanvas.LineTo 将直线画到终点。看来从 TQRShape 派生子类是一个好的解决方案。

我们发现用户使用 TQRShape.Shape 属性来控制绘制何种图形。在 IDE 的窗体上拖入一个 TQRShape 控件,双击窗体,开始书写:QRShape1.Shape。看到了吧,IDE 提示 Shape 是 TQRShapeType 类型。在 Object Inspector 中又很容易知道这个类型定义如下：

```
TQRShapeType =  
(qrsRectangle, qrsVertLine, qrsHorLine, qrsCircle,  
 qrsTopAndBottom, qrsRightAndLeft);
```

别无选择,我们肯定是改写这个属性,以让用户可以画出斜线。qrsTopAndBottom 和 qrsRightAndLeft 是画上下、左右边线的,然后这个任务可以交给 Fram.DrawTop 等完成,所以我们在子类完全可以省略它。这样,我们设计新的 Shape 属性类型如下：

```
TQRShapeTypeEx =  
(qrsRectangle, qrsVertLine, qrsHorLine, qrsCircle, qrsLTTToRB,  
 qrsRTToLB);
```

其中 qrsLTTToRB 从左上角画斜线到右下角,qrsRTToLB 从右上角画斜线到左下角。

从前面的“开发图形图像控件”知道,图形图像是在 Paint 方法中绘出的。所以这个控件运行原

理就是在 Paint 方法中根据绘出 Shape 属性指定的图形。

另外这个控件还有特殊之处，那就是绘出的图形还要能打印出来。这段打印程序是不是要我们自己写出来。相信 QuickReport 开发人员没有那么傻吧，如果每个控件都要重写打印程序，那就是非常糟糕的设计，Borland 也不会买他们的东西了。我们用下面一段程序分析一下 TQRShape 的派生关系：

```
var
  Clss: TClass;
  S: String;
begin
  Clss := QRShape1.ClassParent;
  while Clss <> nil do
  begin
    S := S + Clss.ClassName + #13;
    Clss := Clss.ClassParent;
  end;
  ShowMessage(S);
end;
```

看到了吧，TQRShape 派生于 TQRPrintable 类。很显然，通用打印程序在 TQRPrintable 中已经实现了。在 TQRPrintable 的子类中，只要覆盖 Print 方法将图形图像绘到打印画布上就可以了。

也就是说，Paint 和 Print 方法没有本质区别，除了它们绘制到不同画布外。Paint 对应的画布是 TQRShape.Canvas，Print 对应的画布是 TQRShape.QRPrinter.Canvas。Print 方法的原型是这样的：

```
procedure Print(OfsX, OfsY: Integer);
```

其中(OfsX, OfsY)表示相对坐标原点。Paint 使用的 Canvas 和控件表面是重合的，其原点就是控件的左上角，即(0,0)。而 Print 是在整个大的画布上绘制所有控件的图形图像，所以必须在画某个控件的图形图像时，必须首先计算出控件所处的位置，进而确定绘画时的参照点，当然这个计算过程是由 TQRPrintable 完成的。

分析到这里，我们可以设计如下子类：

```
type
  TQRShapeTypeEx = (qrsRectangle, qrsVertLine, qrsHorLine, qrsCircle,
    qrsLTToRB, qrsRTToLB);

  TlxpQRShape = class(TQRShape)
  private
    FShape: TQRShapeTypeEx;
```

```

procedure SetShape(Value: TQRShapeTypeEx);
procedure DrawShapes(FCanvas: TCanvas; OfsX, OfsY: Integer);
    {Paint 和 Print 方法都要调用它}
protected
    procedure Paint; override;
    procedure Print(OfsX, OfsY: Integer); override;
published
    property Shape: TQRShapeTypeEx read FShape write SetShape;
    {重新声明属性 Shape}
end;

```

各方法的实现就很简单了，大家可以参看后面的源代码。

注意：该控件在设计时不能透明，总是要占一个矩形区间，所以可能覆盖其他控件，但是在打印预览和打印时是透明的，只会占用它该占的地盘。

以下是控件的完整源代码：

```

unit lxpQRShape;

interface

uses
    Windows, Graphics, SysUtils, Classes, Controls, QuickRpt, QRCtrls;

type
    TQRShapeTypeEx = (qrsRectangle, qrsVertLine, qrsHorLine, qrsCircle,
        qrsLTToRB, qrsRTToLB);

    TlxpQRShape = class(TQRShape)
    private
        FShape: TQRShapeTypeEx;
        procedure SetShape(Value: TQRShapeTypeEx);
        procedure DrawShapes(FCanvas: TCanvas; OfsX, OfsY: Integer);
    protected
        procedure Paint; override;    {Paint 实现预览}
        procedure Print(OfsX, OfsY: Integer); override;
        {Print 实现实际打印, 和预览相比, Print 关键是用 OfsX 和 OfsY 指定图像的原点偏移}
    published
        property Shape: TQRShapeTypeEx read FShape write SetShape;
    end;

```

6.8 开发 QuickReport 组件



```
    Color := Frame.Color;
    Style := Frame.Style;
end;
with FCanvas do
begin
    X := Pen.Width div 2;
    Y := X;
    W := Width - Pen.Width + 1;
    H := Height - Pen.Width + 1;
    if Pen.Width = 0 then
    begin
        Dec(W);
        Dec(H);
    end;
    if W < H then S := W else S := H;
    HM := Width div 2;
    VM := Height div 2;
end;
end;
procedure MoveToEx(X, Y: Integer);
begin
    FCanvas.MoveTo(X + OfsX, Y + OfsY);
end;
procedure LineToEx(X, Y: Integer);
begin
    FCanvas.LineTo(X + OfsX, Y + OfsY);
end;
begin
    InitCanvas(True);
    with FCanvas do
    begin
        if Frame.DrawTop then
        begin
            MoveToEx(X, Y);
            LineToEx(Width - X, Y);
        end;
        if Frame.DrawLeft then
        begin
            MoveToEx(X, Y);
            LineToEx(X, Height - Y);
        end;
    end;
end;
```

```

if Frame.DrawRight then
begin
    MoveToEx(Width - X - 1, Y);
    LineToEx(Width - X - 1, Height - Y);
end;
if Frame.DrawBottom then
begin
    MoveToEx(X, Height - Y - 1);
    LineToEx(Width - X, Height - Y - 1);
end;
end;

InitCanvas(False);
if FShape = qrsCircle then
begin
    Inc(X, (W - S) div 2);
    Inc(Y, (H - S) div 2);
    W := S;
    H := S;
end;

with FCanvas do
    case FShape of
        qrsRectangle:
            Rectangle(X + OfsX, Y + OfsY, X + W + OfsX, Y + H + OfsY);
        qrsCircle:
            Ellipse(X + OfsX, Y + OfsY, X + W + OfsX, Y + H + OfsY);
        qrsHorLine:
            begin
                MoveToEx(X, VM);
                LineToEx(Width - X, VM);
            end;
        qrsVertLine:
            begin
                MoveToEx(HM, Y);
                LineToEx(HM, Height - Y);
            end;
        qrsLTToRB:
            begin
                MoveToEx(X, Y);
                LineToEx(Width - X, Height - Y);
            end;
    end;

```

```
end;  
qrsRTToLB:  
begin  
  MoveToEx(Width - X, Y);  
  LineToEx(X, Height - Y);  
end;  
end;  
end;  
  
end.
```

小结

本节讲述了如何开发 QuickReport 的斜线控件,希望对扩展 QuickReport 控件能起到抛砖引玉的作用。

开发 QuickReport 控件(也适用于其他打印类控件)的一个重点是覆盖如下方法:

```
procedure Print(OfsX, OfsY: Integer);
```

以实现在打印机画布上的图形图像绘制。

第 7 章 组件开发相关工作

本章将讲述组件开发的一些相关工作，比如编写组件、属性编辑器、创建组件图标等。一顿大餐的最后程序就是来碗面条、喝点粥汤、吃点水果，否则不能称为完美的大餐。同样，一个 VCL 组件，如果没有必要的图标和编辑器，用户也不认为它是一个完美的组件。

7.1 包和包编译指令

包是 Borland 指定格式的动态连接库。编写时是 .dpk (Delphi Package ，相当于包的工程文件)，编译后是 .bpl (Borland Package Library)。包编译后也会生成 .dcp 文件，地位类似于单元 (.pas) 编译生成的 .dcu 文件。

包主要由两部分组成：包含 (contains) 和依赖 (requires)。包含是指包含哪些单元，依赖是指包对其他包 (.dcp) 的依赖。给包添加单元时，如果添加单元不包含依赖包，那么 IDE 会提示你需要新的依赖包；如果依赖包不完全，则编译不会通过，这时候你可以手工添加需要的依赖包。

打开一个 .dpk 时，选择菜单 Project|Options|Description 可以看到包编译指令选择。下面用表格分别介绍：

Description

类 别	对应指令	意 义	默 认
Description	{ \$DESCRIPTION 'text' }	生成包在安装后的描述	{ \$DESCRIPTION '' }

Usage options

类 别	对应指令	意 义	默认
Designtime only	{ \$DESIGNONLY on }	包组件在设计时使用，可以安装到组件页	
Runtime only	{ \$RUNONLY on }	不能安装到组件页，只能动态调用	
Designtime or Runtime		可以当作设计时或者运行时包使用	是

Build control

类 别	对应指令	意 义	默 认
Rebuild as needed	{ \$IMPLICITBUILD on }	引用了包单元的应用程序，编译时自动编译引用的包单元	
Explicit rebuild	{ \$IMPLICITBUILD off }	不自动编译。需要你打开包工程编译	是



Package name (只影响 bpl, 而不影响 dpk 和 dcp)

类 别	对应指令	意 义	默 认
LIB Prefix	{ \$LIBPREFIX 'string' }	包名前缀。编译得到 “ string 包名.bpl ”	{ \$LIBPREFIX ‘ ’ }
LIB Version	{ \$LIBVERSION 'string' }	第二扩展名。编译得到 “ 包名.bpl. string ”	{ \$LIBVERSION ‘ ’ }
LIB Suffix	{ \$LIBSUFFIX 'string' }	包名后缀。编译得到“ 包 名 string.bpl ”	{ \$LIBSUFFIX ‘ ’ }

Package name 中 LIB Version 是用得比较多的。如果你的包 YourPack.bpk 需要从 1.0 升级到 2.0, 那么你是否需要建立一个新包呢? 不需要, 在 LIB Version 填写 2.0 就行了, 编译后自动生成 YourPack2.0.bpl。Delphi 的核心包 VCL.bpl 就是这么升级的: VCL50.bpl、VCL60.bpl.....

包编译后可以生成如下文件:

.dcu 每单元一个, 是二进制映像文件。如果是 Linux 平台, 则扩展名为.dpu。

.dcp 一个包只有一个, 包含所有 dcu/dpu 的信息。

.bpl 最终生成的共享库。Linux 平台上为.so。

包被编译后, 会在\$(Delphi)\Projects\Bpl 目录生成对应的.dcp 和.bpl 文件。而.dcu 文件一般生成于.pas 所在目录。当发布一个包时, 发布压缩包中至少应该包含.bpl/.so 和所有.dcu/.dpu 文件。

7.2 创建组件图标

如果一个类在 Delphi 的组件面板已经注册并有一个图标时, 我们从这个组件派生一子组件, 它的图标在默认情况下和父组件相同。如果父类没有图标 (比如父类是 TComponent), 那么默认图标是一个红三角形、蓝正方形和金色圆的组合图案。这时候可以指定一个自定义的图标。该图标应该存在于组件资源文件 (.dcr) 中。

dcr 文件可以使用 Delphi 和 C++Builder 自带的 Image Editor 编辑生成。设生成的 dcr 文件为 OneImage.dcr, 那么我们通过下面的方法可以指定它为一个组件的图标:

- (1) 在 Delphi 中打开包文件.dpk。
- (2) 选择菜单: Project|View Source (查看.dpk 文件的源代码)。
- (3) 在.dpk 源代码中增加: { \$R 'OneImage.dcr' }。最后重新编译包即可。

下面看看如何使用 Image Editor 编辑一个 dcr 文件。

- (1) 运行 Image Editor。
- (2) 选择菜单: File|New|Component Resource File (.dcr)。
- (3) 在弹出窗口中选择 Contents, 在右键弹出菜单中选择 New|Bitmap。

(4) 此时提示设置图标属性。组件图标的大小一般设置为 24×24 或者 32×32 ，颜色设置为 16 色或者 256 色。这时就可以使用 Image Editor 设置的绘图工具编辑图标了，或者从 Photoshop 等图形图像处理工具中复制图形图像。

(5) 保存图标。将 Contents\Bitmap 下的 Bitmap1 改名（方法是使用右键弹出菜单的 Rename 命令）为组件的完整类名（含“T”）。最后选择菜单 File|Save as，文件名设为组件的类名（不含“T”）。

这样就完成一个组件图标编辑了。

7.3 属性编辑器

属性编辑器 (PropertyEditor) 是让用户集中、方便设置组件复杂属性的对话框，是一种特殊组件。如 TButton.Font、TmainMenu.Items 都采用属性编辑器设置属性值。

Delphi 的属性编辑器编写主要由三个单元支持：Delphi\Source\ToolsAPI 下的 DesignIntf.pas、DesignEditors.pas 以及 VCLEditors.pas。DesignIntf 定义了属性编辑器基类：TBasePropertyEditor 和支持接口 IProperty。TBasePropertyEditor 直接派生于 TInterfacedObject，可以简化接口的实现。用户自定义属性编辑器必须是 TBasePropertyEditor 直接或者间接子类并且实现 IProperty 接口。DesignEditors 定义了 TBasePropertyEditor 的直接子类 TPropertyEditor，它同时实现了 IProperty 接口。从 TPropertyEditor 直接或间接派生了大量子类属性编辑器（分布在 DesignEditors 和 VCLEditors 单元），下面逐一作个间接介绍：

TOrdinalProperty	用于有序类型属性，如整数、字符、枚举等。
TIntegerProperty	用于整数类型属性。
TCharProperty	用于字符类型属性。
TEnumProperty	用于枚举类型属性。
TFloatProperty	用于浮点类型属性。
TStringProperty	用于字符串类型属性。
TSetElementProperty	用于可直接修改的集合类型属性。
TSetProperty	用于集合类型属性。
TClassProperty	用于对象类型属性。
TMethodProperty	用于事件类型属性。
TColorProperty	用于颜色类型属性。
TFontCharsetProperty	用于字体字符集类型属性。
TFontNameProperty	用于字体名类型属性。
TFontProperty	用于字体类型属性。

等等，大家也可以在 DesignEditors 和 VCLEditors 单元查看所有的属性编辑器类。

编写属性编辑器时一般从 TPropertyEditor 及其直接或者间接子类派生，可根据具体情况而定。



1. TPropertyEditor 的重要属性和方法

property Designer: IDesigner;

这是 IDE 属性编辑环境的接口，可以通过它和 IDE 交互。

property Value: string;

对应于方法 GetValue 和 SetValue，以字符串形式存取属性编辑器对应属性的值。

procedure Edit; virtual;

当用户双击属性或者点击“...”(如有)要设置属性时，Object Inspector 调用它。比如可以在这里响应用户创建并弹出一个对话框。

function GetAttributes: TPropertyAttributes; virtual;

控制属性编辑器的类型。

TPropertyAttributes 声明如下：

```
TPropertyAttribute = (
  paValueList,      用于枚举属性，在属性右边显示下拉按钮
  paSubProperties,   用于对象属性，在属性下边显示子属性
  paDialog,         属性编辑器是弹出对话框形式，在属性右边显示“...”按钮
  paMultiSelect,    多个具有该属性的组件被同时选择时，显示该属性值
  paAutoUpdate,     在属性值修改过程中就应用修改，而不是所有修改完成后
  paSortList,       排序属性值
  paReadOnly,       属性值不能改变
  paRevertable,     属性值可以被恢复到设置前
  paFullWidthName,  Object Inspector 只显示属性名而不显示属性值
  paVolatileSubProperties, 属性值改变过程中刷新子属性值
  paVCL,            属性编辑器用于 VCL 而不是 CLX
  paNotNestable     在显示还有子属性的属性（如对象）时，不显示此属性
);
```

function GetValue: string; virtual;

将属性值格式化为一个字符串返回。

procedure SetValue(const Value: string); virtual;

应用属性值的改变。

procedure Initialize; override;

初始化属性编辑器。在用户使用属性编辑器前调用。属性编辑器的初始化应该在 Initialize 里而不



是构造函数里，因为属性编辑器很多时候创建了但是并没有使用，比如多选时候。

上面几个方法在子类中如果要使用就应该进行覆盖。

2. 注册属性编辑器

属性编辑器要注册后才能使用。而且其使用和一般组件不同，它是由 Object Inspector 自动创建并调用其方法的。注册语法如下：

```
procedure RegisterPropertyEditor(
  PropertyType: PTypeInfo;    属性类型信息，一般是 TypeInfo (属性类型)
  ComponentClass: TClass;    使用该属性的组件类型，如果是 nil 表示可以用于所有组件
  const PropertyName: string; 属性名，如果为 '' 表示可以用于 PropertyType 指定
                              类型的所有属性。
  EditorClass: TpropertyEditorClass;  属性编辑器类型
);
```

如：

```
RegisterPropertyEditor(TypeInfo(TFileName), nil, '',
  TFileNameProperty);
```

表示所有组件的所有 TFileName 类型属性都使用 TFileNameProperty 属性编辑器。

下面看一个例子，打开光盘的“源代码\第6章”的“lxbuaa.dpk”的单元“DemoEditor”可以看到全部源代码。首先定义一个简单的不可视组件，直接从 TComponent 继承：

```
type
  TFilePathName = type String;
  { 定义一个类型 TFilePathName，其内容是字符串，但是和 String 是两种不同类型。注意：
    如果写成 TFilePathName = String; 那么 TFilePathName 就变成 String 的别名了，
    是相同类型。 }

  TTestEditor = class(TComponent)
  private
    FFilePathName: TFilePathName;
  published
    property FilePathName: TFilePathName read FFilePathName write
      FFilePathName;
  end;
```

它只是简单地发布一个 FilePathName 属性，没有在 TComponent 上扩展任何功能。然后实现一个



TFilePathName 类型的属性编辑器：

```
TFilePathNameEditor = class(TStringProperty)
public
    function GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
end;
```

因为 TFilePathName 实际上是 String 类型，所以直接从 TStringProperty 继承。覆盖两个方法。方法实现如下：

```
{ TFilePathNameEditor}
function TFilePathNameEditor.GetAttributes: TPropertyAttributes;
begin
    inherited GetAttributes;
    Result := [paDialog];      { 指定属性编辑器是对话框形式。 }
end;

procedure TFilePathNameEditor.Edit;
begin
    inherited;
    { 创建一个选择文件对话框，然后返回文件路径和文件名给属性 FilePathName。 }
    with TOpenDialog.Create(nil) do
    begin
        if Execute then
            Value := FileName;
        Free;
    end;
end;
```

最后，注册属性编辑器和组件：

```
procedure Register;
begin
    RegisterComponents('lxpbuaa', [TTestEditor]);
    RegisterPropertyEditor(TypeInfo(TFilePathName), nil, '',
        TFilePathNameEditor);
end;
```



现在来看看我们的劳动成果。File|Close All, 然后新建一个工程, 在“lxbuaa”组件页拖一个 TTestEditor 到 Form1 上, 点击其“FilePathName”属性, 看到 FilePathName 后面出现“...”按钮, 点击按钮, 弹出一个“打开文件”对话框!

注意 DesignIntf.pas、DesignEditors.pas 需要文件 designide.dcp 支持, 所以应将 designide.dcp 加入包 lxbuaa.dpk, 方法是在包界面的节点“Requires”上点击右键, 选择“Add”按钮, 最后选择 Delphi\Lib\designide.dcp 文件。

7.4 组件编辑器

组件编辑器 (ComponentEditor) 可以让用户一次设置该组件的多个甚至全部属性, 可以通过双击组件或者右键菜单调用。如 TTreeView 的 Items Editor, TImageList 的 ImageList Editor。

组件编辑器仍然由 DesignIntf 和 DesignEditors 提供支持。DesignIntf 定义了组件编辑器基类 TBaseComponentEditor 和支持接口 IComponentEditor。TBaseComponentEditor 派生于 TInterfacedObject。用户定义组件编辑器必须直接或者间接派生于 TBaseComponentEditor 并且实现接口 IComponentEditor。DesignEditors 定义了 TBaseComponentEditor 的直接子类 TComponentEditor, 它实现了 IComponentEditor。所以一般从 TComponentEditor 派生自己的组件编辑器。

1. TComponentEditor 的重要属性和方法

```
property Designer: IDesigner;
```

这是 IDE 属性编辑环境的接口, 可以通过它和 IDE 交互。

```
property Component: TComponent;
```

表示属性编辑器正在编辑的组件。

```
procedure Copy; virtual;
```

将组件复制到剪贴板。这是一个虚拟方法, 需要覆盖实现。

```
procedure Edit; virtual;
```

双击组件时调用。Edit 在内部调用 GetVerbCount, 如果返回值大于 0, 那么再调用 ExecuteVerb(0)。

```
procedure ExecuteVerb(Index: Integer); virtual;
```

执行 GetVerb 添加到组件右键菜单中的菜单项的对应任务。选择右键菜单项时调用。

```
function GetVerb(Index: Integer): string; virtual;
```

向右键菜单添加菜单项。在组件上点击右键时调用。

```
function GetVerbCount: Integer; virtual;
```

指定 GetVerb 添加的菜单项数目。

我们知道，在 TButton、TForm 上双击时，会自动产生 OnClick 或者 OnCreate 事件句柄。这是怎么实现的呢？Delphi 从 TComponentEditor 派生了 TDefaultEditor 类，作为所有组件的默认编辑器，如果组件注册了特定编辑器，就调用特定编辑器处理，否则调用默认编辑器。假如调用了默认编辑器，并且组件有事件属性，那么双击组件时会自动产生一个事件句柄。其产生规则是：如果有“OnCreate”、“OnChange”或者“OnClick”，则是三者中最先找到的；如果没有，则是所有事件属性中最先找到的。它提供一个重要的虚拟方法：

```
procedure EditProperty(const Prop: IProperty; var Continue: Boolean);  
virtual;
```

它是由 Edit 方法内部调用，通过覆盖这个方法，可以改变事件句柄的产生规则。例如：

```
procedure TOneComponentEditor.EditProperty(  
    const Prop: IProperty; var Continue: Boolean);  
begin  
    if CompareText(Prop.GetName, 'OnMethod2') = 0 then  
        { "OnMethod2" 是组件TComponent_ComponentEditor 定义的一个事件属性。这样覆  
          盖以后，双击TComponent_ComponentEditor会产生OnMethod2句柄而不是其他。}  
        inherited EditProperty(Prop, Continue);  
end;
```

要注意的是，除非你想要改变默认事件句柄，否则组件编辑器应该从 TComponentEditor 继承；因为从 TDefaultEditor 派生时，双击组件会生成事件句柄，不再具有右键菜单第一项的能力了。

2. 注册组件编辑器

```
procedure RegisterComponentEditor(  
    ComponentClass: TComponentClass;      使用该组件编辑器的组件类  
    ComponentEditor: TComponentEditorClass); 组件编辑器类
```

下面看个例子。该例子和上面的属性编辑器例子在同一个单元。我们定义：

```
TOneComponentEditor = class(TComponentEditor)  
public  
    function GetVerb(Index: Integer): string; override;  
    function GetVerbCount: Integer; override;
```

```
procedure ExecuteVerb(Index: Integer); override;
end;
```

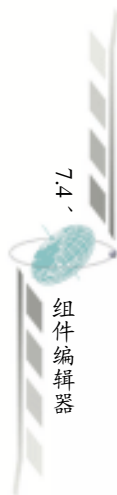
实现：

```
{ TOneComponentEditor}
procedure TOneComponentEditor.ExecuteVerb(Index: Integer);
var
    S: String;
    I: Integer;
begin
    inherited;
    with Component as TTestEditor do
    begin
        S := FilePathName + ';' + Name;
        S := InputBox('输入框', '请输入 FilePathName、Name 属性值, 用 ; 隔开', S);
        I := Pos(';', S);
        FilePathName := System.Copy(S, 1, I-1);
        Name := System.Copy(S, I+1, Length(S));
    end;
    Designer.Modified; {通知 IDE 刷新属性值。}
end;

function TOneComponentEditor.GetVerb(Index: Integer): string;
begin
    Result := '组件编辑器';
end;

function TTestEditor.GetVerbCount: Integer;
begin
    Result := 1;
end;
```

上面代码的意思是：双击组件或者选择右键菜单的“组件编辑器”时，执行 ExecuteVerb。ExecuteVerb 中显示一个对话框，让用户修改组件的 FilePathName 和 Name 属性，并将修改应用到组件。这只是为了说明原理，使用中，一般设计一个窗体，在 ExecuteVerb 中显示给用户。由于我们只添加一个菜单项，所以在 GetVerb 和 ExecuteVerb 中没有用到 Index 参数。如果添加多个，在 GetVerb 应该这样写：





```
case Index of
  0: {}
  1: {}
  .....
end;
```

相应地，在 ExecuteVerb 中应该判断 Index，以响应不同的菜单项任务。

最后是注册：

```
RegisterComponentEditor(TTestEditor, TComponentEditor);
```

将 TComponentEditor 这个组件编辑器注册给 TTestEditor 组件使用。



第 8 章 常用函数和过程

妈妈做菜时常常抱怨菜刀不好使或者什么调味品又不够味了。因此，选择最好的菜刀和最佳的调料是至关重要的。有现成的好东西时，没必要再自己去造。客人已经在客厅里坐着等开饭了，我们才去找铁匠打菜刀，显然会让人不高兴。

本章详细归纳了 Delphi 中常用、好用的几类函数和过程，它们就是进行开发时要使用的菜刀和调料。掌握它们的性能并熟练使用它们，可以在开发中起到事半功倍的效果。

8.1 数据类型转化类

本节所列函数和过程一般都定义在 System 或者 SysUtils 单元。

8.1.1 数值和字符串的相互转化

```
procedure Str(X [: Width [: Decimals ]]); var S);
```

将数值 X 按照一定格式转化为字符串 S。Width 指定 S 的总长度，如果 X 是实数，Decimals 指定小数点后的位数。如：

```
var
  S: String;
begin
  Str(12.2:6:2,S); {S=' 12.20'}
end;

procedure Val(S; var V; var Code: Integer);
```

将字符串 S 转化为数值 V。如果不能转化，则 Code 返回第一个非法字符的位置。如：

```
var
  V,Code: Integer;
begin
  Val('12.00', V, Code);{没能成功转化,Code返回'.'的位置3}
end;
```



8.1.2 整数和字符串的相互转化

```
function IntToStr(Value: Integer): string; overload;  
function IntToStr(Value: Int64): string; overload;
```

第一个用于 32Bit 整数的转化，第二个用于 64Bit 整数的转化。

因为 Cardinal、Longword 等和 Integer 是兼容的（即是说一个 Integer 变量总是可以无损失地存储一个 Cardinal 或者 Longword 变量的值，因为 Integer 的取值范围完全包含了它们的取值范围）。

```
function StrToInt(const S: string): Integer;  
function StrToInt64(const S: string): Int64;
```

将一个字符串转化为整数。如果 S 包含非数字字符（如“1A”、“1.2”），则运行时产生异常。但是 S 也可以是十六进制表示方法，如“\$A”，那么会返回十进制数 10。

```
function StrToIntDef(const S: string; Default: Integer): Integer;  
function StrToInt64Def(const S: string; Default: Int64): Int64;
```

转化的时候可以指定默认值。如果 S 非法，那么它们返回 Default 指定的默认值，不会产生异常。

```
function TryStrToInt(const S: string; out Value: Integer): Boolean;  
function TryStrToInt64(const S: string; out Value: Int64): Boolean;
```

转化得到的整数保存在输出参数 Value，如果不能转化，则函数返回 False。

```
function IntToHex(Value: Integer; Digits: Integer): string; overload;  
function IntToHex(Value: Int64; Digits: Integer): string; overload;
```

将十进制整数转化为十六进制整数，结果用字符串表示。

```
function HexToBin(Text, Buffer: PChar; BufSize: Integer): Integer;
```

将十六进制整数转化为二进制整数，结果用字符串表示，保存在 Buffer 中。

8.1.3 实数和字符串的相互转化

```
function FloatToStr(Value: Extended): string;  
function CurrToStr(Value: Currency): string;
```

实数转化为字符串。

```
function FloatToStrF(Value: Extended; Format: TFloatFormat;  
    Precision, Digits: Integer): string;  
function FormatFloat(const Format: string; Value: Extended): string;
```





```
function FloatToText(Buffer: PChar; const Value: TValue; ValueType: TFloatValue;  
    Format: TFloatFormat; Precision, Digits: Integer): Integer;  
function CurrToStrF(Value: Currency; Format: TFloatFormat;  
    Digits: Integer): string;  
function FormatCurr(const Format: string; Value: Currency): string;
```

转化时，可以指定格式。

```
function StrToFloat(const S: string): Extended;  
function StrToCurr(const S: string): Currency;
```

字符串转化为实数。

类似的函数还有：StrToFloatDef、StrToCurrDef、TryStrToFloat、TryStrToCurr 等，大家顾其名即可思其义了。

8.1.4 实数子类型的相互转化

```
function FloatToCurr(const Value: Extended): Currency;
```

等等。

8.1.5 布尔类型和字符串的相互转化

```
function StrToBool(const S: string): Boolean;
```

将字符串转化为布尔值。

```
function BoolToStr(B: Boolean; UseBoolStrs: Boolean = False): string;
```

将布尔值转化为字符串。

8.2 字符串处理类

注意一个字符串的字符索引是从 1 而不是从 0 开始。

```
function Length(S): Integer;
```

返回字符串长度（字节数）。如果 S 是 Unicode 类型，则返回字节数的一半。

```
procedure SetLength(var S; NewLength: Integer);
```

设置一个字符串的长度。Length、SetLength 以及下面讲的 Copy 也可以用于动态数组。用它改变字符串或者动态数组的长度后，原来长度部分的数据能够保留。

```
procedure SetString(var s: string; buffer: PChar; len: Integer);
```





设置一个字符串的内容（从 buffer 复制）和长度。

```
function Pos(Substr: string; S: string): Integer;
```

取得字符串的起始位置。

```
function StringOfChar(Ch: Char; Count: Integer): string;
```

用 Count 个 Ch 合成一个字符串。

```
function DupeString(const AText: string; ACount: Integer): string;
```

用 ACount 个 AText 合成一个字符串。

```
function Concat(s1 [, s2,..., sn]: string): string;
```

合并多个字符串。等价于 $s1 + s2 + \dots + sn$ 。

```
procedure Insert(Source: string; var S: string; Index: Integer);
```

插入子字符串。

```
function Copy(S: string; Index, Count: Integer): string;
```

获取子字符串。它也可用于动态数组。

```
function StringReplace(const S, OldPattern, NewPattern: string;  
  Flags: TReplaceFlags): string;
```

用 NewPattern 替换 S 中的子字符串 OldPattern。

```
procedure Delete(var S: string; Index, Count: Integer);
```

删除字符串的一部分。

```
procedure UniqueString(var str: string); overload;
```

```
procedure UniqueString(var str: WideString); overload;
```

指定字符串只能有一个引用。一般不会用到，除非需要将 str 转化为 PChar 或者 PWideChar 并改变字符串的内容，这时候可以避免不安全引用。

```
function WrapText(const Text, BreakStr: string; nBreakChars: TSysCharSet;  
  MaxCol: Integer): string; overload;
```

```
function WrapText(const Text: string, MaxCol: Integer = 45): string;  
  overload;
```

分割一个字符串。





MaxCol 表示一行的最大长度。

BreakStr 表示在每行末尾要插入的分割符。如果使用第二种格式，则插入#13#10（回车换行符）。它遇到 nBreakChars 的某字符就分割一次。如果使用第二种格式，则检查空格、连字符、跳格符。

```
function QuotedStr(const S: string): string;
```

用单引号将字符串包裹起来。字符串中的引号会被自动处理。给 SQL 语句中的参数赋值时非常有用。

```
function LastDelimiter(const Delimiters, S: string): Integer;
```

Delimiters 是多个字符的组合。它返回某个字符在 S 中最后出现的位置。

```
function IsDelimiter(  
    const Delimiters, S: string; Index: Integer): Boolean;
```

看看 LastDelimiter 就明白了。

```
function CompareStr(const S1, S2: string): Integer;
```

比较两个字符串（会区分大小写）。

```
function SameText(const S1, S2: string): Boolean;  
function CompareText(const S1, S2: string): Integer;
```

比较两个字符串（不区分大小写）。

```
function LeftStr(const AText: string; ACount: Integer): string;
```

取得左边部分字符串。

```
function LowerCase(const S: string): string;
```

将所有字符转化为小写。

```
function UpperCase(const S: string): string;
```

将所有字符转化为大写。

```
function MidStr(  
    const AText: string; const AStart, ACount: Integer): string;
```

取得部分字符串。它等价于 Copy，主要是方便从 VB 转过来的 Delphi 程序员。

```
function ReverseString(const AText: string): string;
```



翻转字符串。

```
function RightStr(const AText: string; ACount: Integer): string;
```

取得右边部分字符串。

```
procedure Str(X [: Width [: Decimals ]]; var S);
```

格式化一个字符串并保存到 S。X 可以是整数或者实数，Width 和 Decimals 是整数，S 是一个字符串或者字符数组。

```
function Trim(const S: string): string; overload;
```

```
function Trim(const S: WideString): WideString; overload;
```

去掉字符串头尾的空格和控制字符（回车（#13）、换行（#10）、跳格（#8））。

类似的还有 TrimLeft、TrimRight。

由于字符串的处理在开发中占用相当重要的地位，所以下面再从两个角度深入介绍字符串。

8.2.1 字符串的分类

在 Delphi 的数据类型中，字符串是非常复杂的一种。这是由两个原因引起的：

（1）Pascal 的 ShortString 和其他很多语言在实现上有不同。在 C/C++ 中，字符串是 null-terminated 的，即以 #0（空字符）结束，而 ShortString 是用字符串的第一个字节（0）保存字符串的长度。这样就涉及一个相互转化的问题。例如，在 Delphi 中调用 Win32API 时，你不可能直接传递一个 ShortString 字符串，因为 API 函数在这样一个字符串中找不到 #0，不知道它在哪里结束。幸运的是，Delphi 中可以直接在 Pascal 字符串和 null-terminated 字符串间简单转化：PChar(String), String(PChar)。而且 String(PChar) 可以由编译器自动实现，如：可以直接按值传递一个 PChar 变量给一个 String 类型参数。

（2）多字符集的问题，这是所有语言都要面对的。在英文中，一个 8 位的数据就可以表示所有字符了，而对于中文等亚洲文字呢？汉字有几万个！表示这样的字符就必须用 16 甚至更多位数据了。于是各种字符集应运而生。从表示一个字符需要的数据字节数来看，字符集可以划分为以下三种：单字节字符集（SBCS），如 ANSI；双字节字符集（DBCS），如 Unicode；多字节字符集（MBCS），现在还没有广泛应用，但是不排除将来使用的可能。

Delphi 用 String 来处理单字节字符串，用 WideString 来处理双字节字符串（也不排除将来处理多字节字符串的可能）。这个分类的意思不是说 String 只能容纳英文字符，而 WideString 只能处理汉字等字符，而是说 String 和 WideString 在处理字符串的内容时，分别按单字节和双字节进行。比如：

```
var
  S: String;
  WS: WideString;
```

```

begin
  S := '1 个汉字';
  WS := S;
  ShowMessage('S 长度: '+IntToStr(Length(S)));      {7}
  ShowMessage('WS 长度: '+IntToStr(Length(WS)));    {4}
end;

```

String 分为两种：ShortString 和 AnsiString。

ShortString 固定长度为 256 字节（其中第一字节（0）用来存储长度，所以实际只有 255 个字节供用户使用）。它是在低版本 Pascal 中使用的，所以在新的开发中不再使用。

AnsiString 也称为长字符串，它是一个指针，指向一块动态分配的内存。动态分配是按照字符串的长度进行的。它最大可以达到 2 吉字节长，所以我们可以认为它可以存储的字符长度只是跟可能获取的最大内存量有关系。因为它是变长的，所以不再使用第一个字节存储长度。

对于一个 String 类型标识符，编译器自动转化为 ShortString 或者 AnsiString。这和一个编译指令 {\$H} 有关系，当 {\$H+}（默认）时，表示 AnsiString，否则是 ShortString。String[Integer] 可以用来定义一个固定长度但不等于 256 的 ShortString 变量。

对一个单字节和多字节混用的字符串操作时，必须注意不要将一个多字节字符斩断。为此，Delphi 中定义了以“Ansi”开头的过程和函数，专门处理字节混用的字符串。比如：

```

var
  S: String;
begin
  S := '123 汉字';
  ShowMessage(S[Length(S)]); {本来想取出"字"，但是这个代码却取出了"字"的第
                             二个字节}
  ShowMessage(AnsiLastChar(S)); {这句就对了，取出了"字"}
end;

```

在这种情况下，也可以使用 WideString。如上面将 S 声明为 WideString，那么 S[Length(S)] 的值也是“字”。但是使用 WideString 有一个副作用，那就是 Win95/98 的 API 中用于处理 WideString 的函数很少，因而 VCL 如果需要调用 API 对 WideString 处理时，还要经过复杂的转化。

8.2.2 和字符串相关的类

TStrings (Classes 单元)

TStrings 是字符串列表对象的基类，它也是一个抽象类，一般不要直接创建它的实例。它主要提供三种功能：

- (1) 对指定位置的字符串进行管理。



(2) 在文件和流上进行字符串列表操作。

(3) 将字符串和对象关联。

TStrings 的重要属性，如：

```
property CommaText: string;
```

按照 SDF (system data format) 格式，在字符串列表、字符串之间转换。

字符串列表 字符串。将列表中所有字符串转化为一个字符串。转化规则是：

(1) 所有字符串被逗号分割。

(2) 如果字符串中含有空格、逗号、双引号，则该字符串头尾会加上双引号。

(3) 字符串中的双引号被自动重复。

如：

Str,ng 1

Str"ng 2

String 3

String4

返回：

"Str,ng 1","Str"ng 2","String 3",String4

对于字符串 字符串列表：

(1) 不在双引号中的空格、逗号是分割符。

(2) 连续的逗号分割符会生成空字符串。

(3) 逗号分隔符前后的空格分割符不作为分割符，被忽略。

如：

"Str,ng 1","Str"ng 2", , String 3,String4

返回：

Str,ng 1

Str"ng 2

(空字符串)

String

3

String4

```
property DelimitedText: string;
```

按照 Delimiter 和 QuoteChar 在字符串列表、字符串之间的转换。当 Delimiter 为逗号，QuoteChar



是双引号时，DelimitedText 和 CommaText 完全一样。

```
property Delimiter: Char;  
property QuoteChar: Char;
```

参看 DelimitedText。

```
property Text: string;
```

在列表各项的末尾加上回车换行符 (#13#10) 进行字符串列表、字符串之间的转换。

```
property Names[Index: Integer]: string;
```

当列表中字符串是 “Name=Value” 格式时，得到 Name。它是只读属性。

```
property Values[const Name: string]: string;
```

当列表中字符串是 “Name=Value” 格式时，得到 Value。

Names 和 Values 属性在类 TValueListEditor 中大量用到。

```
property Strings[Index: Integer]: string; default;
```

引用第 Index 项字符串。

```
property Objects[Index: Integer]: TObject;
```

引用第 Index 项字符串关联的对象。

TStrings 的重点方法：

```
function AddObject(const S: string; AObject: TObject): Integer; virtual;
```

将一个具有关联对象的字符串增加到列表项中，返回其放置的位置。TStrings 不会拥有该对象，只有指针引用。

```
procedure AddStrings(Strings: TStrings); virtual;
```

增加一个字符串列表的所有字符串到末尾。

```
function Equals(Strings: TStrings): Boolean;
```

比较两个列表。它不比较字符串关联的对象。

```
procedure Exchange(Index1, Index2: Integer); virtual;
```

交换两个项的位置。

```
procedure LoadFromFile(const FileName: string); virtual;
```

从文件读取列表项。文件中每行对应列表中每项。

```
procedure SaveToFile(const FileName: string); virtual;
```

将列表项保存到文件。文件中每行对应列表中每项。

```
procedure LoadFromStream(Stream: TStream); virtual;
```

将流读到 Text 属性，然后转化为列表项。回车符 (#13) 或换行符 (#10) 或二者的任意组合作为分割符。

```
procedure SaveToStream(Stream: TStream); virtual;
```

将 Text 属性的值保存到流。

```
procedure Move(CurIndex, NewIndex: Integer); virtual;
```

将 CurIndex 处列表项移到 NewIndex 处。

TStringList (Classes 单元)

TStringList 是 TString 的直接子类。它实现了 TString 的所有抽象方法，并且增加了新的功能，所以一般用 TStringList 构造字符串列表实例。新增功能包括：

- (1) 对列表项排序。
- (2) 可以阻止重复的列表项被加入。
- (3) 列表项变化时，有事件可以响应。
- (4) 可以指定是否区分大小写。

TStringList 的重要属性：

```
property CaseSensitive: Boolean;
```

是否区分大小写。

```
property Sorted: Boolean;
```

是否对列表项排序，默认为 False。为 True 时，按升序排列，相当于调用方法 Sort。

```
property Duplicates: TDuplicates;
```

当列表被升序排列（包括按升序插入、Sorted=True 或者调用 Sort 方法等）后，用户试图插入和已有列表项重复的字符串时作何处理。

```
type TDuplicates = (dupIgnore, dupAccept, dupError);
```

分别是不能插入、允许插入、不能插入且触发异常。

TStringList 的重点方法：

```
procedure Sort; virtual;
```

将列表项按升序排列。

```
function Find(const S: string; var Index: Integer): Boolean; virtual;
```

当列表被升序排列（包括按升序插入、Sorted=True 或者调用 Sort 方法等，在非升序状态下，该方法总是返回 False）后，查找 S 在列表中是否存在，Index 返回位置。一般不使用这个方法，而使用 IndexOf。

```
procedure CustomSort(Compare: TStringListSortCompare); virtual; 其中
```

```
type TStringListSortCompare = function(
```

```
  List: TStringList; Index1, Index2: Integer): Integer;
```

自定义排序方法。规则如下：

如果 Index1 处字符串应该在 Index2 之前，应返回负数；

如果相等，应返回 0；

如果应该在 Index2 之后，应返回正数。

TStringList 有两个事件：

```
property OnChange: TNotifyEvent;
```

列表变化后触发。

```
property OnChanging: TNotifyEvent;
```

列表变化前触发。

THashedStringList (IniFiles 单元)

它直接派生于 TStringList，没有增加任何新的属性、方法、事件，只是在内部使用了哈希表，加快了列表项搜索的速度。如果列表项数目很大时，则使用它可以显著提高处理速度。

8.3 流处理类

```
procedure ObjectBinaryToText(Input, Output: TStream); overload;
```

```
procedure ObjectBinaryToText(Input, Output: TStream;
```

```
var OriginalFormat: TStreamOriginalFormat); overload;
```

将二进制流（保存在 Input 中）转化为可读文本流（Output）。OriginalFormat 指定 Input 中数据的原始格式：二进制还是文本。

TStream.WriteComponent 可创建实例的二进制流。

```
procedure ObjectResourceToText(Input, Output: TStream); overload;  
procedure ObjectResourceToText(Input, Output: TStream;  
  var OriginalFormat: TStreamOriginalFormat); overload;
```

将资源流（保存在 Input 中）转化为可读文本流（Output）。OriginalFormat 指定 Input 中数据的原始格式：二进制还是文本。

TStream.WriteComponentRes 或者 TResourceStream 可创建资源流。

对应上面两个过程的反过程是 ObjectTextToBinary 和 ObjectTextToResource。

```
function ReadComponentRes(  
  const ResName: string; Instance: TComponent): TComponent;  
function ReadComponentResEx(  
  HInstance: THandle; const ResName: string): TComponent;  
function ReadComponentResFile(  
  const FileName: string; Instance: TComponent): TComponent;
```

从资源中（通过存储资源的模块句柄、资源名、资源文件名）读取实例数据。

```
procedure WriteComponentResFile(  
  const FileName: string; Instance: TComponent);
```

将实例数据按照资源格式保存到文件。

```
function TestStreamFormat(Stream: TStream): TStreamOriginalFormat;  
测定流的格式，可能返回 sofUnknown、sofBinary、sofText。
```

由于流的处理在开发中占有相当重要的地位，所以下面再深入介绍流及其相关的类。

TStream (Classes 单元)

TStream 是所有流对象的基类，直接从 TObject 继承。它是一个抽象类，不要直接创建它的实例。它内部使用 TFile 的两个子类：TReader 和 TWriter。

TStream 的重要属性：

```
property Position: Int64;
```

被存取流的当前位置。内部调用 Seek 方法。



```
property Size: Int64;
```

流的大小。内部调用 Seek 和 SetSize。

TStream 的重要方法：

```
function CopyFrom(Source: TStream; Count: Int64): Int64;
```

从流 Source 复制数据到当前流。Count 表示复制量。CopyFrom 从 Source.Position 位置开始复制，如果你需要复制整个 Source，应该在调用 CopyFrom 之前设置：Source.Position := 0。

```
procedure ReadBuffer(var Buffer; Count: Longint);
```

从当前流的 Position 位置读取 Count 长度数据到缓冲区 Buffer。Buffer 一般是无类型指针(Pointer) 或者有类型指针(PChar、PInteger 等，取决于数据的具体类型)。它内部调用 Read 方法。

```
procedure WriteBuffer(const Buffer; Count: Longint);
```

在当前流的 Position 位置后插入 Count 长度数据，数据源是 Buffer。它内部调用 Write 方法。

```
function Read(var Buffer; Count: Longint): Longint; virtual; abstract;  
function Write(const Buffer; Count: Longint): Longint; virtual; abstract;
```

大家看到 Read 和 Write 是两个抽象方法，它们需要由子类实现。同时，它们可以返回本次操作实际作用的数据量，这个返回值小于或者等于 Count。

```
function Seek(Offset: Longint; Origin: Word): Longint; overload; virtual;  
function Seek(const Offset: Int64; Origin: TSeekOrigin): Int64; overload;  
    virtual;
```

改变当前 Position。第一个用于 32 位流，第二个用于 64 位流。没有实现，子类至少需要覆盖其中之一。

```
procedure SetSize(NewSize: Longint); overload; virtual;  
procedure SetSize(const NewSize: Int64); overload; virtual;
```

改变流的大小。

```
function ReadComponent(Instance: TComponent): TComponent;
```

用流中数据初始化实例。如果 Instance=nil，那么它构造一个实例。

```
procedure WriteComponent(Instance: TComponent);
```

将实例数据保存到流。





```
function ReadComponentRes(Instance: TComponent): TComponent;
procedure WriteComponentRes(
  const ResName: string; Instance: TComponent);
```

它们和上面两个方法类似，如果流是 Windows 资源文件格式（比如子类 TResourceStream 使用），应该使用这两种方法。

下面看几个例子：

(1) 如何读取对象数据并让我看到它是什么样？

```
procedure TForm1.Button1Click(Sender: TObject);
var
  BinStrm: TMemoryStream;    { 内存流，用它来读出对象数据 }
  StrStrm: TStringStream;    { 字符串流，用来得到对象数据的字符串格式 }
begin
  BinStrm := TMemoryStream.Create;
  StrStrm := TStringStream.Create('');
  BinStrm.WriteComponent(Button1);          { 读入 Button1 的数据 }
  BinStrm.Position := 0;
  ObjectBinaryToText(BinStrm, StrStrm);    { 将二进制数据转化为字符串 }
  StrStrm.Position := 0;
  ShowMessage(StrStrm.DataString);
  FreeAndNil(BinStrm);
  FreeAndNil(StrStrm);
end;
```

(2) 如何从流中还原组件实例？

```
procedure TForm1.Button2Click(Sender: TObject);
var
  FileStrm: TFileStream;
  StrStrm: TStringStream;
  BinStrm: TMemoryStream;
  Button: TButton;
begin
  FileStrm := TFileStream.Create('ReadButton.txt', fmOpenRead);
  { 用文件保存 Button 的数据 }
  StrStrm := TStringStream.Create('');
  BinStrm := TMemoryStream.Create;
  Button := TButton.Create(Self);

  FileStrm.Position := 0;
```

```

StrStrm.CopyFrom(FileStrm, FileStrm.Size);    { 复制到中间流 }
StrStrm.Position := 0;
ObjectTextToBinary(StrStrm, BinStrm);        { 复制到目的流 }
BinStrm.Position := 0;
BinStrm.ReadComponent(TComponent(Button));    { 从流中读出 Button }
Form1.InsertControl(Button);                  { 在 Form1 上显示 Button }
FreeAndNil(FileStrm);
FreeAndNil(StrStrm);
FreeAndNil(BinStrm);
end;

```

实际上，Delphi 的 IDE 就是使用上面两种技术把界面上的组件保存到 dfm，然后再从 dfm 中读出组件，还原到界面上。

(3) 再讲一个例子，看如何使用 TStream 的 ReadBuffer 和 WriteBuffer 方法。

```

procedure TForm1.Button3Click(Sender: TObject);
var
    FileStrm: TFileStream;
    I: Integer;
    Ps: PChar;
begin
    FileStrm := TFileStream.Create('WriteFile.txt', fmCreate);
    for I := 0 to Memo1.Lines.Count-1 do
        begin
            Ps := PChar(Memo1.Lines[I]);
            FileStrm.WriteBuffer(Ps^, Length(Ps));    { 有类型指针 }
            { 或 FileStrm.WriteBuffer(Pointer(Memo1.Lines[I])^, Length(Ps));
              无类型指针 }
        end;
    FreeAndNil(FileStrm);
end;

```

这里只是为了演示如何用 WriteBuffer 写数据。其实 Memo1.Lines 是 TStrings 类型，而 TStrings.SaveToFile 方法可以直接写文件（读文件是 LoadFromFile）。但是 SaveToFile 内部也是用 TFileStream 实现的，请看它的源代码：

```

procedure TStrings.SaveToFile(const FileName: string);
var
    Stream: TStream;

```




```
begin
  Stream := TFileStream.Create(Filename, fmCreate);
  try
    SaveToStream(Stream);
  finally
    Stream.Free;
  end;
end;
```

TStream 的重要子类：

TStringStream	存取字符串
TBlobStream	存取 Blob 字段
TWinSocketStream	从 Socket 连接存取数据
TOleStream	使用 COM 接口存取数据
THandleStream	和有 Windows 句柄的对象交互，如 FileOpen 得到的是一个有句柄文件
TFileStream	存取文件（派生于 THandleStream）
//TCustomMemoryStream	存取内存数据的抽象类，派生以下两个类
TMemoryStream	存取内存缓冲数据
TResourceStream	存取资源

8.4 内存管理、程序流程控制类

8.4.1 内存管理

```
function AllocMem(Size: Cardinal): Pointer;
```

在堆中分配一块大小为 Size 的内存，将其内容初始化为 0，并返回它的地址指针。如果无须初始化，可以使用另一个过程：

```
procedure GetMem(var P: Pointer; Size: Integer);
```

AllocMem 实际上是这样实现的：

```
function AllocMem(Size: Cardinal): Pointer;
begin
  GetMem(Result, Size);
  FillChar(Result^, Size, 0);
end;
```

释放 AllocMem 或者 GetMem 分配的内存可用：

```
procedure FreeMem(var P: Pointer[; Size: Integer]);
```

注意：FreeMem 不会将 P 置为 nil。

如果发现先前用 AllocMem 或者 GetMem 分配的内存块大小需要调整，可以使用：

```
procedure ReallocMem(var P: Pointer; Size: Integer);
```

当：

(1) P=nil、Size=0 时，不执行任何操作。

(2) P=nil、Size>0 时，等价于 GetMem。

(3) P<>nil、Size=0 时，等价于 FreeMem。

(4) P<>nil、Size>0 时，重新分配内存。先前的数据不会遭到破坏。如果 P 指向一个动态数组，等价于 SetLength。

```
procedure New(var P: Pointer);
```

它和 GetMem 功能相似。但做了一个附加工作：如果 P 指向一个包含 long strings、variants、dynamic arrays 或者 interfaces 类型字段的结构，那么它会将 long strings 初始化为空，variants、dynamic arrays 和 interfaces 初始化为 Unassigned。初始化工作由：

```
procedure Initialize(var V [ ; Count: Integer ] );
```

在内部完成。

New 对应的释放方法是：

```
procedure Dispose(var P: Pointer);
```

它和 FreeMem 功能相似，不过它会首先将 long strings、variants、dynamic arrays 置为空、interfaces 为 Unassigned。此工作由：

```
procedure Finalize( var V [ ; Count: Integer ] );
```

在内部完成。

因此，如果 P 包含上面所述类型的字段，并且用 AllocMem 或者 GetMem 分配内存后，应该调用 Initialize(P)；在使用 FreeMem 释放前应该调用 Finalize(P)。不过这时候使用 New、Dispose 要方便一些。

8.4.2 程序流程控制

```
procedure Abort;
```

引发一个无提示异常（也称静态异常）。一般地，Abort 会导致退出当前过程或者函数，和 Exit 相似。如果 Abort 位于一个 try...except/finally...end 块中，它将执行流程引到 end 后的语句。注意它和 Exit 是有区别的，Exit 表示退出当前过程或者函数，而 Abort 是引发一个异常，因此会终止当前程序段的执行。例如：

```
procedure DoSomething;
begin
```



```
Abort;  
//Exit;  
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    DoSomething;  
    ShowMessage('OK'); {这句不会被执行。如果是使用Exit而不是Abort,则可以执行}  
end;  
  
procedure AddTerminateProc(TermProc: TTerminateProc);
```

其中

```
TTerminateProc = function: Boolean;
```

将一个 TTerminateProc 类型的函数增加到应用程序结束时要执行的过程列表中。TermProc 的返回值 True/False 决定应用程序是否可以结束。

例如：

```
function TermProc: Boolean;  
begin  
    Result := False;  
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    AddTerminateProc(TermProc);  
end;
```

还有一个过程：

```
procedure AddExitProc(Proc: TProcedure);
```

功能和 AddTerminateProc 是一样的，但它是为保持向后兼容提供的，不要再使用它。

8.5 文件操作类

8.5.1 使用文件句柄进行 I/O 处理

```
function FileCreate(const FileName: string): Integer;
```

创建文件。



```
function FileOpen(const FileName: string; Mode: LongWord): Integer;
```

打开文件。

```
function FileSeek(Handle, Offset, Origin: Integer): Integer; overload;
```

```
function FileSeek(Handle: Integer; const Offset: Int64;
```

```
    Origin: Integer): Int64; overload;
```

移动文件指针。

```
function FileRead(Handle: Integer; var Buffer; Count: Integer): Integer;
```

读文件。

```
function FileWrite(Handle: Integer; const Buffer; Count: Integer):
```

```
    Integer;
```

写文件。

```
procedure FileClose(Handle: Integer);
```

完成读写，关闭文件句柄。

8.5.2 使用 Pascal 文件变量进行 I/O 处理

```
procedure AssignFile(var F; FileName: string);
```

其中 F 是任何类型的文件变量。文件分为类型文件和无类型文件。AssignFile 将一个文件变量和文件名关联。早期版本的 Delphi 中使用 Assign 过程。

类型文件变量：

```
type fileTypeNames = file of DataType
```

```
var DataFile: fileTypeNames;
```

DataType 可以是任何定长的类型，如 Integer、Boolean、定长 string（如 string[20]）、定长 record 等等，而不能是长字符串、Variant、TObject、动态数组、指针等等。

TextFile 是 Delphi 内定的类型文件类型，用于操作文本文件。

无类型文件变量：

```
var DataFile: file;
```

```
procedure Rewrite(var F: File [; RecSize: Word ] );
```

创建文件。RecSize 只是在操作无类型文件变量时有效，用于指定每次文件操作时的数据量，默认是 128 字节。





```
procedure Reset(var F [ : File; RecSize: Word ] );
```

打开文件。

```
procedure Append(var F: TextFile);
```

打开文本文件，追加数据。

```
procedure Rename(var F; Newname: string);
```

重命名文件。

```
function FileSize(var F): Integer;
```

文件中包含多少 DataType 长度的数据。它不能用于 TextFile。

```
*function FileSizeByName(sFilename: string): cardinal;
```

以字节数返回文件数据量。等于 TFileStream.Size。这个函数位于 IdGlobal 单元。IdGlobal 单元主要为 Indy 组件使用，它定义了一些和 Delphi 其他单元名字、目的相同但是实现和返回值不尽相同的过程和函数，所以我们一般不要使用这个单元的过程和函数。

```
function FilePos(var F): Longint;
```

当前文件指针位置。它不能用于 TextFile。

```
procedure Seek(var F; N: Longint);
```

将文件指针从当前位置移动 N 个 DataType 长度。

```
procedure Read(F , V1 [, V2,...,Vn ] : DataType);
```

从类型文件中读取数据到对应类型的变量中。

```
procedure Write(F , V1 [, V2,...,Vn ] : DataType);
```

将对应类型的数据写到类型文件中。

```
procedure Readln([ var F: TextFile; ] V1 [, V2, ...,Vn ]);
```

从文本文件逐行读取数据。如果没有 F 参数，表示从命令行读取。

```
procedure Writeln([ var F: Text; ] P1 [, P2, ...,Pn ] );
```

向文本文件逐行写入数据。

```
procedure BlockRead(var F: File; var Buf;
```



```
Count: Integer [; var AmtTransferred: Integer]);
```

从无类型文件读取 Count 个 RecSize 长度的数据到 Buf 中。AmtTransferred 返回实际读到的数据量 (小于或者等于 Count)。

```
procedure BlockWrite(var f: File; var Buf;
  Count: Integer [; var AmtTransferred: Integer]);
```

向无类型文件写入 Count 个 RecSize 长度的数据。AmtTransferred 返回实际写入的数据量 (等于或者小于 Count; 如遇到磁盘空间不足等情况会出现小于的情况)。

```
procedure CloseFile(var F);
```

完成读写, 关闭文件变量。

8.5.3 面向对象文件 I/O 处理

使用 TFileStream。大家看看帮助就明白了, 很好用。

在文件 I/O 时, 我们推荐使用 Pascal 文件变量和面向对象方法处理。

8.5.4 文件属性操作

```
function FileGetAttr(const FileName: string): Integer;
```

取得文件属性。返回值类型为 TSearchRec. Attr。

```
function FileSetAttr(const FileName: string; Attr: Integer): Integer;
```

设置文件属性。

```
function FileIsReadOnly(const FileName: string): Boolean;
```

文件是否有只读属性。

```
function FileSetReadOnly(const FileName: string; ReadOnly: Boolean):
  Boolean;
```

设置文件是否有只读属性。

```
function FileGetDate(FileHandle: Integer): Integer;
```

返回文件的时间 (修改) 属性。返回值可以用 FileDateToDateTime 转化。

```
function FileAge(const FileName: string): Integer;
```

和 FileGetDate 类似。



```
function FileSetDate(Handle: Integer; Age: Integer): Integer; overload;  
function FileSetDate(const FileName: string; Age: Integer): Integer;  
    overload;
```

设置文件的时间（修改）属性。Age 可用 DateTimeToFileDate 转化得到。

8.5.5 其他函数和方法

```
function DirectoryExists(const Directory: string): Boolean;
```

目录是否存在。

```
function ForceDirectories(const Dir: string): Boolean;  
function CreateDir(const Dir: string): Boolean;
```

创建目录。如果 Dir 中要求的父目录不存在，ForceDirectories 会同时创建。当然你别希望它们创建驱动器名，比如：ForceDirectories('Z:\AA.Txt');是不可能成功的。

```
function RemoveDir(const Dir: string): Boolean;
```

删除空目录。

```
function GetCurrentDir: string;
```

返回当前目录。

```
function SetCurrentDir(const Dir: string): Boolean;
```

设置当前目录。

ChDir、MkDir、Rmdir 等几个过程是为了对应命令行控制而设置的，也可以实现上面几个函数的功能，但是没有异常处理，不推荐使用它们。

```
function SelectDirectory(const Caption: string; const Root: WideString;  
    out Directory: string): Boolean; overload;  
function SelectDirectory(var Directory: string; Options: TSelectDirOpts;  
    HelpCtx: Longint): Boolean; overload;
```

弹出对话框让用户选择工作目录。

```
function ExtractFileDrive(const FileName: string): string;
```

得到驱动器名。

```
function ExtractFilePath(const FileName: string): string;  
function ExtractFileDir(const FileName: string): string;
```





得到路径名。ExtractFilePath 返回的路径最后带 “\”，而 ExtractFileDir 不带。

```
function ExtractRelativePath(const BaseName, DestName: string): string;
```

得到相对路径。

```
function IncludeTrailingPathDelimiter(const S: string): string;
```

确保路径以 ‘\’ 结束。

```
function ExtractFileName(const FileName: string): string;
```

得到文件名。

```
function ExtractFileExt(const FileName: string): string;
```

得到文件扩展名 (含 ‘.’)。

```
procedure ProcessPath (const EditText: string; var Drive: Char;  
    var DirPart: string; var FilePart: string);
```

一次得到驱动器名、路径名、文件名。

```
function RenameFile(const OldName, NewName: string): Boolean;
```

重命名目录或者文件。

```
function ChangeFileExt(const FileName, Extension: string): string;
```

重命名文件扩展名。Extension 应该包含 ‘.’。

```
function DeleteFile(const FileName: string): Boolean;
```

删除文件。

```
function FileSearch(const Name, DirList: string): string;
```

在指定目录搜索文件。DirList 可以用分号分隔的多个目录。返回在该文件所在目录名。

```
function FindFirst(const Path: string; Attr: Integer; var F: TSearchRec):  
    Integer;
```

```
function FindNext(var F: TSearchRec): Integer;
```

用于在指定目录搜索文件或者子目录。

使用这两个函数要注意的是，路径最后应该包含 “*.*”，如：F:\Dir*.*，否则搜索不会成功。具体使用方法可以参考 Delphi 在线帮助。





```
function FileExists(const FileName: string): Boolean;
```

文件是否存在。

文件复制、移动可用 API 函数 CopyFile、MoveFile。其中 MoveFile 可以移动目录。更复杂的操作可以调用 API 函数 SHFileOperation。

8.6 日期时间类

此类函数和过程主要定义在 SysUtils 和 DateUtils 两个单元。

8.6.1 获取/合成日期/时间

```
function Now: TDateTime;
```

返回当前日期和时间。

```
function Date: TDateTime;
```

返回当前日期。

```
function Time: TDateTime;
```

返回当前时间。

```
function YearOf(const AValue: TDateTime): Word;
```

返回指定日期/时间的年份。类似的有：

```
MonthOf, WeekOf, DayOf, HourOf, MinuteOf, SecondOf, MilliSecondOf.
```

如果要一次取得其中的多项，可以使用：

```
procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word);
procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word);
procedure DecodeDateTime(DateTime: TDateTime; var Year, Month, Day, Hour,
    Minute, Second, MilliSecond: Word);
```

如果要合成一个日期/时间，可以使用：

```
function EncodeDate(Year, Month, Day: Word): TDateTime;
function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime;
function EncodeDateTime(Year, Month, Day, Hour, Minute, Second,
    MilliSecond: Word): TDateTime;
```



```
procedure ReplaceDate(var DateTime: TDateTime; const NewDate: TDateTime);
```

更新 DateTime 的日期部分。

```
procedure ReplaceTime(
```

```
  var DateTime: TDateTime; const NewTime: TDateTime);
```

更新 DateTime 的时间部分。类似的有：

RecodeYear/Month/Day 等等可以更新年份、月份、天等数据。

8.6.2 日期/时间和字符串的转换

```
function DateToStr(Date: TDateTime): string;  
function TimeToStr(Time: TDateTime): string;  
function DateTimeToStr(DateTime: TDateTime): string;  
function StrToDate(const S: string): TDateTime;  
function StrToTime(const S: string): TDateTime;  
function StrToDateTime(const S: string): TDateTime;
```

8.6.3 日期/时间的运算

日期/时间数据在本质上是一个浮点数，在 System 单元可以看到：

```
TDateTime = type Double;
```

也就是说，TDateTime 不过是 Double 的别名。该数据的整数 1 表示一天，如：

```
var  
  dt: TDateTime;  
begin  
  dt := Now;      { 现在 }  
  dt := dt + 1;    { 明天 }  
  dt := dt + 0.25; { 6 小时后，因为 6 小时为 0.25 天 }  
end;
```

VCL 提供以下函数和过程封装了这些原始运算：

```
function IncYear(const AValue: TDateTime;  
  const ANumberOfYears: Integer = 1): TDateTime;
```

对年份运算。类似的有 IncMonth、IncDay 等。ANumberOfYears 可以是正数或负数。

```
function WeeksInYear(const AValue: TDateTime): Word;
```

一年中有多少周。



```
function DaysInYear(const AValue: TDateTime): Word;
```

一年中有多少天。

```
function IsInLeapYear(const AValue: TDateTime): Boolean;或者
```

```
function IsLeapYear(Year: Word): Boolean;
```

是否是闰年。

```
function YearSpan(const ANow, AThen: TDateTime): Double;
```

两个日期相差多少年。它假设一年为 365.25 天。

```
function YearsBetween(const ANow, AThen: TDateTime): Integer;
```

它是 Trunc(YearSpan) (直接取整数, 不四舍五入) 的结果。

类似的有 MonthSpan/sBetween、DaySpan/sBetween、WeekSpan/sBetween、HourSpan/sBetween 等。

```
function SystemTimeToDateTime(const SystemTime: TSystemTime):  
TDateTime;
```

将一个系统类型时间转化为 TDateTime 类型。

可用 API 函数 GetSystemTime(SystemTime: PSystemTime)取得系统时间。

8.7 VCL 类

8.7.1 Classes 单元

```
function AllocateHWnd(Method: TWndMethod): HWND;
```

其中 TWndMethod = **procedure**(**var** Message: TMessage) **of** object;

创建一个不可见窗口, Method 过程处理这个窗口的消息。这样创建的窗口没有一个对应的控件。VCL 中很典型的例子是 TTimer, 它使用了这个函数用以处理 WM_TIMER 消息。Method 被称为窗口过程, TControl.WindowProc 就是 TWndMethod 类型, 是 TControl 的窗口过程。

析构这个窗口可使用:

```
procedure DeallocateHWnd(Wnd: HWND);
```

```
function CountGenerations(Ancessor, Descendant: TClass): Integer;
```

获取两个类的继承层次。比如: 如果 Descendant 直接继承于 Ancessor, 那么返回 1。如果 Ancessor 和 Descendant 是相同类, 则返回 0, 如果不相干, 则返回-1。

```
procedure RegisterComponents(  
  const Page: string; ComponentClasses: array of TComponentClass);
```

注册组件，并显示到组件页上。在 procedure Register 中调用。

```
procedure RegisterNoIcon(ComponentClasses: array of TComponentClass);
```

注册组件，但不显示在组件页。在 procedure Register 中调用。

```
procedure RegisterNonActiveX(ComponentClasses: array of TComponentClass;  
  AxRegType: TActiveXRegType);
```

使该类/甚至其子类不能被 ActiveX wizard 直接转化为 ActiveX 控件类。在 procedure Register 中调用。

```
procedure RegisterFields(const FieldClasses: array of TFieldClass);
```

注册字段类，以便 Dataset designer 使用。这个过程定义在 DB 单元。

```
procedure RegisterClass(AClass: TPersistentClass);  
procedure RegisterClasses(AClasses: array of TPersistentClass);  
procedure RegisterClassAlias(  
  AClass: TPersistentClass; const Alias: string);
```

注册类，以便 GetClass、FindClass 和 ReadComponentResFile 等检索。它们一般在 initialization 部分调用，然后在 finalization 部分调用 UnregisterClass、UnregisterClasses 进行注销。

```
procedure RegisterIntegerConsts(AIntegerType: Pointer;  
  AIdentToInt: TIdentToInt; AIntToIdent: TIntToIdent);
```

这个过程讲起来有点麻烦。先不管它，让我们说点别的。

大家知道 TColor 变量的值在内部是用整数表示的，但是在 Object Inspector 中设置的其实是字符串（如“clWindow”）。这就需要一种在大量字符串和数值之间转化的方法。RegisterIntegerConsts 就将这种方法（AIdentToInt 和 AIntToIdent）注册给 AIntegerType（如 TypeInfo（TColor）），变量需要转化时自动调用。

其中：

```
type TIdentToInt = function(const Ident: string; var Int: Longint):  
  Boolean;  
type TIntToIdent = function(Int: Longint; var Ident: string): Boolean;
```

Ident 是 Identifier 的简写。AIdentToInt 和 AIntToIdent 是两个函数类型变量，它们在字符串和对应整数之间作转化。这种转化需要另一个类型的支持：

```
Array of TIdentMapEntry.
```



其中：

```
type
  TIdentMapEntry = record
    Value: Integer;
    Name: String;
  end;
```

其实转化也很简单,就是在这个数组中遍历 Name 或者 Value 找到对应的 Value 或者 Name。Delphi 定义了 IdentToInt 和 IntToIdent 方法实现这个遍历,所以你直接在 AIdentToInt 和 AIntToIdent 中调用它们就行了。注销使用 UnregisterIntegerConsts。

8.7.2 Controls 单元

```
function FindControl(Handle: HWND): TWinControl;
```

找到指定句柄对应的窗口控件。

API 函数 WindowFromDC 可以找到和对应设备环境句柄联系的窗口的句柄。

```
function FindVCLWindow(const Pos: TPoint): TWinControl;
```

找到指定位置的窗口控件。

API 函数 WindowFromPoint 可以找到指定位置的窗口的句柄,进而用 ChildWindowFromPoint 可以找它的子窗口。

```
function GetCaptureControl: TControl;
```

找到当前处理所有鼠标信息的控件。可以用全局变量 Mouse 引用鼠标。

```
procedure SetCaptureControl(Control: TControl);
```

设置一个控件去处理所有鼠标信息。

8.7.3 Dialogs 单元

```
function CreateMessageDialog(const Msg: string; DlgType: TMsgDlgType;
  Buttons: TMsgDlgButtons): TForm;
```

创建用户自定义对话框,需要调用其方法 ShowModal 显示它。

显示全功能对话框可以使用 MessageDlg、MessageDlgPos、MessageDlgPosHelp 等。

显示简单对话框可使用 ShowMessage、ShowMessageFmt、ShowMessagePos。

```
function RemoteLoginDialog(var AUserName, Apassword: string): Boolean;
function LoginDialog(const ADatabaseName: string;
  var AUserName, APassword: string): Boolean;
```



显示一个登录对话框。

```
function LoginDialogEx(const ADatabaseName: string; var AUserName,  
    APassword: string; NameReadOnly: Boolean): Boolean;
```

显示一个登录对话框，并可以指定用户名是否固定。

以上两个函数位于 DBLogDlg 单元。

```
function InputBox(const ACaption, APrompt, ADefault: string): string;
```

显示一个字符串输入对话框。如果用户没有修改 ADefault (即使点击 Cancele), 就返回 ADefault。

```
function InputQuery(const ACaption, APrompt: string; var Value: string):  
    Boolean;
```

功能和 InputBox 同，但是返回值表示用户点击了 OK 还是 Cancele。

```
function IsAbortResult(const AModalResult: TModalResult): Boolean;
```

将 Form.ShowModal 传递给 AModalResult, 如果 ShowModal 返回 mrAbort 或者 mrCancel, 那么此函数返回 True。

```
function IsAnAllResult(const AModalResult: TModalResult): Boolean;
```

假如 ShowModal 返回 mrAll、mrNoToAll 或者 mrYesToAll, 那么此函数返回 True。

```
function IsNegativeResult(const AModalResult: TModalResult): Boolean;
```

假如 ShowModal 返回 mrNo 或者 mrNoToAll, 那么此函数返回 True。

```
function IsPositiveResult(const AModalResult: TModalResult): Boolean;
```

假如 ShowModal 返回 mrYes、mrOk、mrYesToAll 或者 mrAll, 那么此函数返回 True。

```
function StripAllFromResult(const AModalResult: TModalResult):  
    TModalResult;
```

将 ShowModal 返回值的 All 部分去掉，如：

mrAll	mrOk
mrNoToAll	mrNo
mrYesToAll	mrYes

注意：我们不推荐过多使用 Dialogs 单元中的类、方法，尤其是在编写组件中，因为这个单元相对比较大，编译结果也响应的很胖。类似地，应该尽量避免引用 Forms 等超大单元。





对于显示简单的信息，可以直接调用 API 函数 `MessageBox`、`MessageBoxEx`、`MessageBoxIndirect` 等。

8.8 位运算类

Delphi 中提供了 6 个位运算符：`not`、`and`、`or`、`xor`、`shl`、`shr`。

前四个大家都知道是什么意思了。`shl` 和 `shr` 分别表示左移和右移多少位，相当于乘以或者除以 2 的 N 次方。如：

```
var
    I: Integer;
begin
    I := 1000;
    I := I shl 10;
    {即 I := Round(Ldexp(I, 10)); 或者 I := I * Round(Power(2, 10));}
    {ShowMessage(IntToStr(I));}
end;
```

下面看几个例子：

(1) 判断 Integer 中某位是否是 1：

```
type
    TIntBitPos = 0..31;
    {一个 Integer 共 32 位，从高到低（即从右到左）为 0..31 位}

function GetBitOfInt(I: Integer; BitPos: TIntBitPos): Boolean;
begin
    Result := (I shr BitPos) and $1 = 1;
    {首先右移 BitPos 位，那么原来第 BitPos 就移到了最右边（即最低位），然后和 1 作 and
    运算（即将其他位置 0）。如果该位是 1，则返回 True，否则为 False}
end;
```

(2) 替换一个 Integer 的某位：

```
procedure ReplaceBitOfInt(
    var I: Integer; BitPos: TIntBitPos; NewValue: Boolean);
var
    NewV: Integer;
begin
    {如果替换值和原来相同，那么不作任何操作}
```



```

if GetBitOfInt(I, BitPos) <> NewValue then
begin
    { 构造一个临时变量, 第 BitPos 位为 1, 其余位全为 0 }
    if BitPos = 0 then
        NewV := 1
    else
        NewV := 2 shl (BitPos-1);
        { 如果原来位为 0, 新值为 1, 那么在该位加 1; 否则减 1 }
    if NewValue then
        Inc(I, NewV)      { 或者 I := I or NewV }
    else
        Dec(I, NewV);
    end;
end;

```

(3) 以下是 Windows 单元定义的几个函数, 可以直接调用:

```

{ 用低字节 A 和高字节 B 合成一个 Word (16 位) }
function MakeWord(A, B: Byte): Word;
begin
    Result := A or B shl 8;
end;

{ 用低字 A 和高字 B 合成一个 Longint (即 Integer, 32 位) }
function MakeLong(A, B: Word): Longint;
begin
    Result := A or B shl 16;
end;

{ 取双字的高字 }
function HiWord(L: DWORD): Word;
begin
    Result := L shr 16;
end;

{ 取字的低字节 }
function HiByte(W: Word): Byte;
begin
    Result := W shr 8;
end;

```




(4) 以下是 System 单元定义的几个函数，可直接调用：

```
{取整数(Integer、Cardinal、Word、DWord、Int64、Smallint 等)的低字节}
function Lo(X): Byte;

{取整数(Integer、Cardinal、Word、DWord、Int64、Smallint 等)的高字节}
function Hi(X): Byte;
```

8.9 图形图像类

以下过程和函数大部分定义在 Graphics 单元。

```
function Point(AX, AY: Integer): TPoint;
function SmallPoint(AX, AY: SmallInt): TSmallPoint;
```

构造一个点。

```
function PointsEqual(const P1, P2: TPoint): Boolean; overload;
function PointsEqual(const P1, P2: TSmallPoint): Boolean; overload;
```

两点是否重合。

```
function Rect(Left, Top, Right, Bottom: Integer): TRect; overload;
function Rect(const ATopLeft, ABottomRight: TPoint): TRect; overload;
```

构造一个矩形。

```
function ColorToRGB(Color: TColor): Longint;
```

将 TColor 转化为一个 32 位的 RGB 数值。Longint 即 Integer。可以用 API 函数 GetRValue、GetGValue、GetBValue 取得结果中对应的 R、G、B 三种颜色值。

```
function ColorToString(Color: TColor): string;
```

将 TColor 转化为字符串，如 'clBlack'（假如 Delphi 定义了该颜色对应的字符串）或者 '\$02FF8800'。

```
function GetDefFontCharSet: TFontCharset;
```

取得系统缺省字符集。

```
function GraphicExtension(GraphicClass: TGraphicClass): string;
```

取得图形对象类的默认文件扩展名。TBitmap : bmp、TIcon : ico、TMetafile : emf。



```
procedure Frame3D(Canvas: TCanvas; var Rect: TRect;  
  TopColor, BottomColor: TColor; Width: Integer);
```

这个过程位于 ExtCtrls 单元。

它给 Rect 区域画出一个 3D 的外框。Width 指定框的宽度。如果需要产生升起效果，则 TopColor（用在左上部分）应该是亮色，BottomColor（用在右下部分）是暗色。如果需要产生按下效果，则使用相反的颜色。

```
procedure MoveWindowOrg(DC: HDC; DX, DY: Integer);
```

这个过程位于 Controls 单元。

改变 HDC 的原点。一般可用 TCanvas.Handle 或者 API 函数 GetDCEx、GetDCEx 取得 DC。



第 9 章 高级开发技巧

星级大厨都有引以为豪的绝技，没听说只会做回锅肉的人可以独步厨坛。

在本章，让我们来掌握一些实践证明行之有效的技艺，以期提升我们的做菜能力。只有经历这个阶段，做出的菜才能上档次、上规模。至于什么“已经失传多年、近日重现江湖”的、旁门左道的、有重大副作用的（比如要被火燎了胡须）功夫，就算了，我也不会、也不敢学！

9.1 自定义窗口过程

任何一个 TWinControl 都有预定义的窗口过程，可以通过它接收 Application.DispatchMessage 传递的消息。因此，如果我们开发不从 TWinControl 或其子类派生的组件，并希望这个组件能够处理消息，那么必须自定义窗口过程并注册。

请看下面的代码：

```
unit DemoMsgComponent;

interface

uses Windows, Classes, Messages;

type
  TDemoMsgComponent = class(TComponent)
  private
    FHandle: HWND;
    procedure MainWndProc(var Message: TMessage); { 自定义一个窗口过程 }
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Handle: HWND read FHandle; { 发布一个Handle属性,这样别人可以给它发送消息 }
  end;

implementation
```

```

{ TDemoMsgComponent }

constructor TDemoMsgComponent.Create(AOwner: TComponent);
begin
    inherited;
    FHandle := Classes.AllocateHWnd(MainWndProc);
    { AllocateHWnd是Classes单元的全局过程, 需要一个TWndMethod类型的参数( 即一个窗
      口过程); 其作用是创建一个不可见窗口并将参数传入的窗口过程关联于它, 返回所创建窗
      口的句柄}
end;

destructor TDemoMsgComponent.Destroy;
begin
    if FHandle <> 0 then
        Classes.DeallocateHWnd(FHandle);    { 销毁已创建的窗口}
    inherited;
end;

{ 以下就是窗口过程的实现, 这里只是简单地演示接收一个WM_CLOSE消息, 别的调用
  TObject.DefaultHandler进行默认处理}
procedure TDemoMsgComponent.MainWndProc(var Message: TMessage);
begin
    if Message.Msg = WM_CLOSE then
        MessageBox(0, '收到消息WM_CLOSE', '收到消息', MB_OK)
    else
        DefaultHandler(Message);
end;

end.

```

然后使用一段代码就可以演示这个例子了：

```

procedure TForm1.Button1Click(Sender: TObject);
var
    DemoMsgCom: TDemoMsgComponent;
begin
    DemoMsgCom := TDemoMsgComponent.Create(nil);
    SendMessage(DemoMsgCom.Handle, WM_CLOSE, 0, 0);
    FreeAndNil(DemoMsgCom);
end;

```



9.2 自定义消息及其替代方法

我们常常需要在一个 TWinControl 实例的外部设置或者获取它的一些信息。通常情况下，只是知道该 TWinControl 实例的句柄（Handle 属性）。此时，常用方法是给它发送特定消息，如以下可以取得一个 TEdit 实例的 Text 属性值：

```
var
  P: PChar;
begin
  GetMem(P, MAXBYTE);           {MAXBYTE是定义在Windows单元的一个常数，等于255}
  SendMessage(Edit1.Handle, WM_GETTEXT, MAXBYTE, Integer(P));
  { Integer(P) 也可以改为 :PInteger(@P)^。表示变量P的地址，P用来存放取得的Text值。}
  ShowMessage(P);
  FreeMem(P);
end;
```

如果使用标准的 Windows 和 VCL 定义的消息还无法满足我们的要求，那么可以自定义特定消息。比如：

```
const MY_MESSAGE = WM_USER + 1;

.....

procedure DoMY_MESSAGE(var Msg: TMessage); message MY_MESSAGE;

.....

procedure TForm1.DoMY_MESSAGE(var Msg: TMessage);
begin
  ShowMessage(Name);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  SendMessage(Handle, MY_MESSAGE, 0, 0);
end;
```

WM_USER 是 Messages 单元定义的常数，等于\$0400（\$表示十六进制数据）。在一个类中，自定



义消息必须在 WM_USER 到 0x7FFF 范围 (VCL 控件自定义的消息就是位于这个范围), 因为其他范围的数值是为操作系统和其他一些特殊操作预留的。

大家对自定义消息的方法可能比较熟悉了, 在这里我不多占篇幅讲解。下面主要介绍一种新的方法来实现类似功能, 那就是存取属性。

```

.....

published
  property OneProperty: Integer read FOneProperty write SetOneProperty;
end;

function SetOneProp(Handle: HWND; PropName: String; PropValue: Integer):
  Boolean;

.....

uses TypInfo; { TypInfo 包含了相关运行时类型信息的大量函数、过程和常数 }

.....

{ 以下这个函数是用来在知道一个实例的Handle情况下, 设置它的某个属性的值 }
function SetOneProp(Handle: HWND; PropName: String; PropValue: Integer):
  Boolean;
var
  tfContrl: TWinControl;
begin
  tfContrl := FindControl(Handle);
  { FindControl 是 Controls 单元的一个函数, 它根据 Handle 返回对应的 TWinControl 实例。以下设置一个控件属性的方法, 大家可以参考 "运行时类型信息" 一小节的内容。 }
  Result := (tfContrl <> nil) and (GetPropInfo(tfContrl, PropName) <> nil);
  if Result then
    SetOrdProp(tfContrl, PropName, PropValue);
end;

procedure TForm1.SetOneProperty(Value: Integer);
begin
  FOneProperty := Value;
  case FOneProperty of
    0:
      ShowMessage(Name);
  end;
end;

```



```

    end;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    SetOneProp(Handle, 'OneProperty', 0);
end;

```

从上面可见,对于只知道一个实例的 Handle,而需要设置或者获取它某些信息时,使用自定义消息和存取属性两种方法,有异曲同工之妙。

9.3 自定义系统惟一消息

在“自定义消息及其替代方法”中已经讲了在 WM_USER 到 0x7FFF 范围自定义消息。这个“自定义”有两个层次的意思:

- (1) 控件提供者定义的消息。如 VCL 已有控件定义了一些消息。
- (2) 控件使用者扩展控件时定义的消息。

所以,WM_USER 到 0x7FFF 范围的消息在不同的类中使用,即使常数值相等,含义也是不同的。换句话说,不同的类中可以使用相同的常数来定义不同的消息。

如果要定义系统惟一的,即在这台计算机中,再没有第二个类或者应用程序使用相同的常数作为一个消息,那么就不能简单地指定 WM_USER+N 来定义一个消息,而应该使用 API 函数来让系统返回一个未使用的常数来代表一个消息。这个 API 函数是:

```

UINT RegisterWindowMessage(
    LPCTSTR lpString { address of message string });

```

其中参数 lpString 是一个 PChar 兼容类型的字符串。这个字符串代表该消息的名字。

如果使用相同的 lpString 参数多次调用 RegisterWindowMessage (即使是在不同的应用程序中调用),那么每次返回的常数都是相同的。这就保证了该消息常数的惟一性,同时,不同的类实例和不同的应用程序也可以使用这个惟一的常数来传递相同的消息。

要求在不同应用程序中传递自定义消息时,常常需要使用这个函数来生成一个消息常数。

下面来看个例子。我们使用两个应用程序,每个都只有一个窗体。先看接受端的代码:

```

type
    TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    private
        ID: DWORD;
    { 我们分别使用 Application 和 Application.MainForm 来接受别的应用程序发送的消

```

```

    息}
    {使用Application.OnMessage事件来处理消息}
    procedure NewOnMessage(var Msg: TMsg; var Handled: Boolean);
protected
    {覆盖Application.MainForm.WndProc来处理消息}
    procedure WndProc(var Message: TMessage); override;
end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

{ TForm1 }

procedure TForm1.FormCreate(Sender: TObject);
begin
    Caption := '接受者';
    ID := RegisterWindowMessage(PChar('OneOfAppMessage'));
    {获取一个系统唯一的消息常数}
    Application.OnMessage := NewOnMessage;
end;

procedure TForm1.NewOnMessage(var Msg: TMsg; var Handled: Boolean);
begin
    if Msg.message = ID then
    begin
        ShowMessage('应用程序收到消息');
        Handled := True;
    end;
end;

procedure TForm1.WndProc(var Message: TMessage);
begin
    inherited;
    with Message do if Msg = ID then
    begin
        ShowMessage('主窗体收到消息');
        Result := 0;
    end;
end;

```




```
end
end;
```

接下来是消息发送端的代码：

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    ID: DWORD;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  ID := RegisterWindowMessage(PChar('OneOfAppMessage'));
  { 获取一个系统唯一的消息常数，它和接受端注册的消息常数是相同的 }
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  H: HWND;
begin
  H := FindWindow(nil, PChar('接受者')); { 找到接受端的主窗体的句柄 }
  if H <> 0 then
  begin
    SendMessage(H, ID, 0, 0); { 将消息发送给接受端的主窗体 }
    PostThreadMessage(GetWindowThreadProcessId(H), ID, 0, 0);
    { 将消息发送给接受应用程序。注意给应用程序发送消息应该使用API函数
      PostThreadMessage。 }
  end;
end;
```



```
{GetWindowThreadProcessId通过主窗体的句柄来取得应用程序句柄}
end;
end;
```

9.4 新颖的类工厂

在 COM 编程中，一个 COM 类(CoClass)必须有至少一个类工厂(class factory，实际上 Delphi 中只有惟一的一个类工厂，用来创建所有注册了的 COM 类实例)和惟一的类标示(CLSID)。CoClass 实例由指定的类工厂根据传递的 CLSID 参数创建。

在非 COM 开发中，有时候也具有类似的要求。比如我的工程包含 10 个 Form，对应某个界面上的 10 个选项，我希望用户选定某项后，创建对应的 Form 并显示出来。这时候，我非常渴望用一个过程完成这项工作，我希望只告诉它是哪个选项就行。你现在能做到吗？如果能，那么不用看下面了；不能，请听我下面的分解。

首先给大家引见 Delphi 中一个很特别的类型：类引用类型。

```
class of type
```

其中 type 可以是任何一个普通类型。如：

```
type
  TClass = class of TObject;
  TPersistentClass = class of TPersistent;
  TComponentClass = class of TComponent;
  TFormClass = class of TForm;
```

等等，都是 Delphi 内部已经定义的类型引用类型。你也可以定义任意的类引用类型。类引用类型用来干什么？

(1) 可以当作对应的普通类来定义变量，如：

```
type
  Obj1: TObject;
  Obj2: TClass;
```

这两种定义是完全一样的。

(2) 在方法参数中传递类而不是类实例，从而使参数可以调用类方法。这是最主要的。如：

```
function TForm1.MyGetClassName(AnyClass: TClass): String;
begin
  Result := AnyClass.ClassName;
```



```
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    ShowMessage(MyGetClassName(TForm1));  
end;
```

在方法的参数中，类引用类型和类两种参数(如 TClass 和 TObject)的关系如同类和对象的关系。

构造函数是一种特殊的类方法。借助上面讲的类引用类型，可以实现一个非常特殊的构造函数：新颖的类工厂。之所以“新颖”，是因为类工厂是 COM 中的概念，所以我在这里借用它。这个类工厂可以根据传入的类引用类型参数创建对应的类实例。如：

```
function TForm1.ClassFactoryForTWinControl(  
    AnyTWinControlClass: TWinControlClass;  
    AOwner: TComponent): TWinControl;  
begin  
    Result := AnyTWinControlClass.Create(AOwner);  
end;  
  
procedure TForm1.Button2Click(Sender: TObject);  
begin  
    with ClassFactoryForTWinControl(TForm2, Application) do  
    begin  
        ShowMessage(Name);  
        Width := 100;  
        Height := 100;  
        Left := 10;  
        Show;  
    end;  
end;
```

其中 TWinControlClass 是 Delphi 内部定义的：TWinControlClass = class of TWinControl。TForm 是 TWinControl 的子类，所以 TForm 的子类 TForm2 可以作为 TWinControlClass 类型的参数传入。

怎么样，你那 10 个 Form 的问题解决了吧！

下面再讲讲这个类工厂的扩展功能。我可不可以传递像类名这样一个字符串而不是类引用类型，也让它构造出我请求的对象？当然可以！这个时候，我们的类工厂必须实现一个新机制：根据某个类名字符串找到对应的类引用类型。

Delphi 的 Classes 单元提供这样的功能！很简单的函数：



```
function FindClass(const ClassName: string): TPersistentClass;
```

和：

```
function GetClass(const ClassName: string): TPersistentClass;
```

其中 ClassName 是完整的类名。

它们的区别只有一个：如果找不到结果，FindClass 要触发异常，而 GetClass 不会。当然找不到时都返回 nil。

好，我们试试：

```
ShowMessage(GetClass('TForm1').ClassName);
```

嗯，为什么发生内存读错误？你不是说 GetClass 不会触发异常吗？呵，GetClass('TForm1')不会有错，但是它返回 nil，你还调用 nil.ClassName，那就是你的不对了。不过 TForm1 明明是存在的，为什么 GetClass 返回 nil 呢？

这里我要告诉大家，GetClass 和 FindClass 跟你所说的 TForm1 存不存在毫无关系。它们的参数对应的类引用都必须注册，否则就找不到。这是由它们的实现机制决定的。这个机制实现在 Classes 单元中，整个过程比较复杂，简单的可以这么描述它：

Classes 内部定义了一个 TList 类型的变量 FClassList。大家知道 TList 可以管理任何类型的指针(当然包括类引用)，你注册一个类引用时，也就是将它增加到 FClassList 里，注销一个类引用时，FClassList 会删除这个项。然后，GetClass 和 FindClass 就在 FClassList 里面找类引用。怎么找呢？它遍历 FClassList，判断当前项的 ClassName 是不是等于你传过来的类名，找到了就返回；找完了都没有就只好返回 nil 了。所以你不注册就找不到，除非“别人”已经帮你注册。

注册需要用到 Classes 单元的两个过程：

```
procedure RegisterClass(AClass: TPersistentClass);  
procedure RegisterClasses(AClasses: array of TPersistentClass);
```

使用格式如下：

```
RegisterClass(TForm1);  
RegisterClasses ([TForm1, TForm2]);
```

不使用时应该注销，用两个过程：

```
procedure UnRegisterClass(AClass: TPersistentClass);  
procedure UnRegisterClasses(AClasses: array of TPersistentClass);
```

你可以在任意地方调用这四个过程，但是标准的格式是这样的：



```
{ 在单元加载时注册}
initialization
  RegisterClasses ([TForm1, TForm2]);
{ 在单元销毁时注销}
finalization
  UnRegisterClasses ([TForm1, TForm2]);
```

好，让我们按照上面所说的定义一个 ClassFactoryForTWinControl 的重载方法：

```
function ClassFactoryForTWinControl(
  AnyTWinControlClass: TWinControlClass;
  AOwner: TComponent): TWinControl; overload;

function ClassFactoryForTWinControl(
  AnyTWinControlClassName: String;
  AOwner: TComponent): TWinControl; overload;

function TForm1.ClassFactoryForTWinControl(
  AnyTWinControlClassName: String;
  AOwner: TComponent): TWinControl;
var
  FindClass: TPersistentClass;
begin
  FindClass := GetClass(AnyTWinControlClassName);
  if (FindClass <> nil) and FindClass.InheritsFrom(TWinControl) then
    Result := TWinControlClass(FindClass).Create(AOwner)
  else Result := nil;
end;
```

调用它：

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  with ClassFactoryForTWinControl('TForm2', Application) do
  begin
    ShowMessage(Name);
    Width := 100;
    Height := 100;
    Left := 10;
    Show;
  end;
end;
```



哇，ClassFactoryForTWinControl(TForm2', Application)和 ClassFactoryForTWinControl(TForm2, Application)的调用结果完全一样！

不过在实际应用中，上面的类工厂还不够完善。如果用户请求的对象在以前已经创建过，那么应该返回先前创建的那个对象，而不是另起炉灶重新创建一个吧。所以实用的类工厂中，也应该具有类似上面所说的“注册”功能，已经创建的直接返回它，没有创建的才重新创建。我们引入 Delphi 中一个十分重要的辅助类：TObjectList(Containers 单元)。顾名思义，它用来管理一个对象数组。

TObjectList 管理对象有两种方式：(1) 引用管理；(2) 拥有管理。采用哪种管理方式由一个属性决定：

```
property OwnsObjects: Boolean;
```

OwnsObjects 默认为 True，在 True 状态下，TObjectList 所有拥有对象，即拥有管理方式：

- (1) Extract/ Delete、Remove、Assign 一个项的同时会析构对应的对象。
- (2) Clear 会析构所有拥有的对象。
- (3) TObjectList 析构时会析构所有拥有的对象。

由于我们只是用来记录请求的对象是否创建过，所以采用引用方式。下面开始具体的编码。

再声明一个重载方法：

```
function ClassFactoryForTWinControl(  
    AnyTWinControlClassName: String;  
    AOwner: TComponent;  
    NewOrRunning: Boolean): TWinControl; overload;
```

其中 NewOrRunning 为 True 时，表示可以返回已经创建过的对象，如果为 False，则总是创建新对象。

实现：

```
function TForm1.ClassFactoryForTWinControl(  
    AnyTWinControlClassName: String;  
    AOwner: TComponent; NewOrRunning: Boolean): TWinControl;  
var  
    FindPos: Integer;  
    FindClass: TPersistentClass;  
    CreateNew: Boolean;  
begin  
    FindClass := GetClass(AnyTWinControlClassName);  
    if (FindClass = nil) or (not FindClass.InheritsFrom(TWinControl)) then  
    begin  
        Result := nil;  
        Exit;
```



```

end;

if NewOrRunning then
begin
    { 注意下面的FindInstanceOf。FindInstanceOf在ObjList中找对应类型的类实例，
      如果该对象已经析构，则返回-1表示没找到(即使对象指针不为nil，如使用Free而不是FreeAndNil)。如果调用了FreeAndNil(Form1)，那么Form1=nil，但是ObjList
      中对应项指针并不等于nil，因为它和Form1是两个指向了同一个对象的不同指针。}
    FindPos := ObjList.FindInstanceOf(FindClass);
    CreateNew := FindPos = -1;
end else
    CreateNew := True;
if CreateNew then
begin
    Result := TWinControlClass(FindClass).Create(AOwner);
    { 如果需要创建新对象，我们应该将创建的对象指针加入ObjList，即"注册" }
    ObjList.Add(Result);
end else { 如果不需要创建新对象，直接返回已创建的对象 }
    Result := TWinControl(ObjList[FindPos]);
end;

```

ObjList 的管理如下：

```

type
    .....
private
    ObjList: TObjectList;

procedure TForm1.FormCreate(Sender: TObject);
begin
    {TObjectList实现了重载构造函数：
    (1) 按默认OwnsObjects属性为True创建：
    constructor Create; overload;
    (2) 可以自行设定OwnsObjects属性：
    constructor Create(AOwnsObjects: Boolean); overload;
    我们需要OwnsObjects=False，所以调用第二个构造函数}
    ObjList := TObjectList.Create(False);
end;

```



```
procedure TForm1.FormDestroy(Sender: TObject);
begin
    FreeAndNil(ObjList);
end;
```

这个函数的调用如下：

```
procedure TForm1.Button4Click(Sender: TObject);
begin
    { 如果第三个参数为False，那么调用效果和Button3Click一样。连续点击Button4，我们
      的方法创建了TForm2的多个实例。 }
    with ClassFactoryForTwinControl('TForm2', Application, True) do
    begin
        ShowMessage(Name);
        Width := 100;
        Height := 100;
        Left := 10;
        Show;
    end;
end;
```

9.5 使用对象库

运行 Delphi，选择菜单“File|New|Other”，你看到的窗体里包含了大量预定义对象，这就是 Delphi 的对象库(Object Repository)。通过使用对象库，Delphi 让你以十分简洁的步骤创建一些实际很复杂的类和对象，如 COM 对象、数据模块、Web 程序等等。

我们常常将集中的功能写入组件、将常用函数和过程写在一个单元，通过使用组件和共享单元来简化软件开发。但是似乎只有很少的人使用对象库。对象库是个好东西。比如一个项目中有很多类似的窗体时，我们可以将这个窗体抽象后放入对象库，然后在软件中需要的地方从对象库提取这个窗体，可以大大加快开发速度、并使以后的修改非常方便。

1. 将窗体加入对象库

在窗体上点击右键，选择“Add to Repository”，然后根据提示填写相应项，这样就将一个窗体加入对象库了。

2. 从对象库提取窗体

选择菜单“File|New|Other”，翻到对象所在页，选中需要的对象。这时候你看到有选项“Copy、Inherit、Use”。对于“Data Modules、Forms、Dialogs”这三页下的对象，三个选项都是可能的。那么到底该选择哪一个呢？这首先需要我们搞清楚它们到底是什么意思。

Copy：将原对象复制一个新对象到工程中。修改原对象或者工程中的对象互不影响。



Inherit：从原对象继承一个新对象到工程中。新对象上的组件/控件不能删除，但可以修改属性(比如不需要一个 Button，那么将它 Visible 置为 False 即可)。对原对象的修改会反映到新对象。工程中会同时包含原对象和新对象。

Use：直接使用原对象。这时候不会像 Copy 或者 Inherit 那样生成新对象，所以修改这个加到工程项目中的对象就是修改原对象。

很显然，Inherit 给我们提供了非常灵活和强大的后续处理手段和能力。因此，工程需要使用对象库中的对象时，常常也是以 Inherit 为提取方式。

下面看一个具体的例子。

我们现在要开发一个 MIS 系统，根据业务需要，使用 20 个窗体对应数据库中的 20 个表，要求用户能通过这 20 个窗体对对应 20 个表的数据进行浏览、增加、修改、删除。你会不会总是使用“File|New|Form”来创建这 20 个窗体呢。如果是，恭喜你，现在给你一个新方法，它会免去你很多烦恼。到目前为止，你大概给 20 个窗体都编写了增加、修改、删除代码(它们其实都是相似的，有区别的大概就是数据表名和数据集组件(TTable /TQuery，TADOTable/TADOQuery))，因为区别不大，所以你写好一个窗体的代码后，复制到其他 19 个窗体然后做了一些表名、数据集不同的修改。当某一天你突然发现有一句代码有点问题，好，修改 20 个窗体；改完了，却发现还有一点问题……这时相信你会很痛苦，你在想，有什么办法可以只改一处，其他自动修改？

我们来使用对象库的方法：

首先，我们创建一个抽象于 20 个窗体的 Form。这个 Form 应该实现 20 个窗体通用的功能(要依据窗体的数据才能完全实现的功能，要在抽象层次实现、留下接口)。窗体大致如图 9-1 所示。

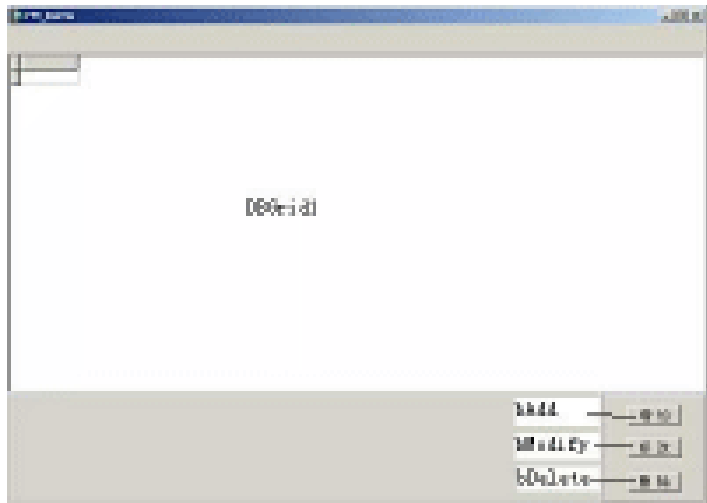


图 9-1 对象库

代码（因为只是个例子，有些地方做了简化）如下：

```

unit UT_DATA;

interface

uses
  Windows, Classes, Controls, Forms, Buttons, ExtCtrls, StdCtrls, Grids,
  DBGrids, DB, DBTables;

type
  TFM_DATA = class(TForm)
    pEdit: TPanel;
    pDataParent: TPanel;
    pDCParent: TPanel;
    pDC: TPanel;
    pMod: TPanel;
    bAdd: TBitBtn;
    bModify: TBitBtn;
    bDelete: TBitBtn;
    pCommit: TPanel;
    bCommit: TBitBtn;
    bCancle: TBitBtn;
    pData: TPanel;
    DBGrid1: TDBGrid;
    procedure bAddClick(Sender: TObject);
    procedure bModifyClick(Sender: TObject);
    procedure bDeleteClick(Sender: TObject);
    procedure bCommitClick(Sender: TObject);
    procedure bCancleClick(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    thisDataSet: TDataSet;
    procedure SetMode(Mode: Word);
  published
  end;

var

```



```

FM_DATA: TFM_DATA;

implementation

{$R *.dfm}

{ 自定义过程部分 }
procedure TFM_DATA.SetMode(Mode: Word);
begin
    case Mode of
        0:    { 浏览 }
            pMod.BringToFront; { 将有增加、修改、删除按钮的面板pMod显示到前面 }
        1:    { 编辑 }
            pCommit.BringToFront; { 将有确定、取消按钮的面板pCommit显示到前面 }
    end;
end;

{ 事件部分 }
procedure TFM_DATA.bAddClick(Sender: TObject);
begin
    thisDataSet.Append;    { 增加记录 }
    SetMode(1);
end;

procedure TFM_DATA.bModifyClick(Sender: TObject);
begin
    thisDataSet.Edit;      { 编辑记录 }
    SetMode(1);
end;

procedure TFM_DATA.bDeleteClick(Sender: TObject);
begin
    thisDataSet.Delete;    { 删除记录。实际应用中，应该增加“是否删除？”等确认功能 }
end;

procedure TFM_DATA.bCommitClick(Sender: TObject);
var
    DataSetType: Word;
begin
    { 取得数据集的类型代号，以便在下面对不同的数据集使用不同的数据操作方法 }
    if thisDataSet is TTable then DataSetType := 1

```

```

else if thisDataSet is TQuery then DataSetType := 2
else DataSetType := 0;

if DataSetType <> 0 then
with TDBDataSet(thisDataSet).Database do
try
    if not InTransaction then StartTransaction;
    thisDataSet.Post;
    Commit;
    with thisDataSet do
    case DataSetType of
        1: Refresh;
        2:
            begin
                Close;
                Open;
            end;
    end;
except
    Rollback;
end;
end;

procedure TFM_DATA.bCancleClick(Sender: TObject);
begin
    thisDataSet.Cancel;           { 取消数据更新 }
    SetMode(0);
end;

procedure TFM_DATA.FormCreate(Sender: TObject);
begin
    try
        thisDataSet := DBGrid1.DataSource.DataSet;
        { 初始化变量thisDataSet }
        thisDataSet.Open;
    except
        MessageBox(0, '数据源设置不正确。', '提示',
            MB_ICONINFORMATION+MB_OK);
    end;
end;

```



```

procedure TFM_DATA.FormCloseQuery(
  Sender: TObject; var CanClose: Boolean);
begin
  { 如果窗口关闭时数据集处于编辑状态, 那么应该要求用户对此作适当的处理 }
  if thisDataSet.State in [dsEdit, dsInsert] then
    begin
      case MessageBox(0, '保存变动过的数据吗?', '请选择',
        MB_ICONQUESTION+MB_YESNOCANCEL) of
        IDYES: bCommit.Click;
        IDNO: bCancle.Click;
        IDCANCEL: CanClose := False;
      end;
    end;
  end;
end;

end.

```

其中定义了一个很重要的变量：thisDataSet: TDataSet。它在 FormCreate 中被初始化为：
thisDataSet := DBGrid1.DataSource.DataSet。这样，整个窗体中使用 thisDataSet 来代表窗体对应的数据集，从而将浏览、增加、修改、删除等功能完全抽象出来加以实现。从它继承的窗体会自动继承这些功能，不需要为实现这些功能再写任何代码。

然后将它添加到对象库。这时候新建一个工程，提取这个对象（采取 Inherit 方式），则其单元内容如下：

```

.....
TFM_DATA2 = class(TFM_DATA)
.....

```

如果我们不满意 TFM_DATA 中的一些处理代码，我们抛弃重写：

```

procedure TFM_DATA2.bAddClick(Sender: TObject);
begin
  { inherited; 这样就不会执行从TFM_DATA继承来的功能 }
  { 新的代码 }
end;

```

也可以在 TFM_DATA 中定义一些虚拟、动态或者抽象方法，并在一些处理过程中调用，但是这些方法部分甚至完全在子类中实现。

现在我们只需要修改 TFM_DATA 而不需要 20 个窗体！



9.6 非发布 (published) 数据的持久化

Delphi 可以将 published 属性数据存取到 dfm 文件。那么对非 published 属性能否实现同样功能呢？

首先必须明白 Delphi 是如何持久化 published 属性的。在 VCL 构架一小节已经知道 TPersistent 提供流式读写属性的能力：

```
procedure DefineProperties(Filer: TFile); virtual;
```

DefineProperties 是通过调用 TFile 类来实现属性读写的，具体是调用 TFile 的两个方法：

(1) 用于标准类型属性，如整数、布尔、枚举、字符串：

```
procedure DefineProperty(const Name: string; ReadData: TReaderProc;
  WriteData: TWriterProc; HasData: Boolean); virtual; abstract;
```

(2) 用于二进制类型属性，如图像、声音：

```
procedure DefineBinaryProperty(const Name: string; ReadData,
  WriteData: TStreamProc; HasData: Boolean); virtual; abstract;
```

我们通过覆盖这个方法 procedure DefineProperties(Filer: TFile)可以实现非发布数据持久化。

下面的例子演示如何持久化标准类型数据（此例子在 lxpbaa.dpk 包中）：

```
unit Comp_NonePubProps;

interface

uses
  Windows, Messages, SysUtils, Classes;

type
  TComp_NonePubProps = class(TComponent)
  Private
    FStr: String;           { 我们要持久化FStr这个私有字符串 }
    procedure ReadStrData(Reader: TReader);      { 数据读方法 }
    procedure WriteStrData(Writer: TWriter);     { 数据写方法 }
  protected
    { 覆盖TComponent. DefineProperties }
    procedure DefineProperties(Filer: TFile); override;
  public
    constructor Create(AOwner: TComponent); override;
  end;
```



```

implementation

constructor TComp_NonePubProps.Create(AOwner: TComponent);
begin
    inherited;
    FStr := 'FStr';           { 给定FStr默认值 }
end;

procedure TComp_NonePubProps.DefineProperties(Filer: TFile);
begin
    inherited;
    { 随便指定属性名为StrProp; 当FStr<>' '时才进行存取 }
    Filer.DefineProperty('StrProp', ReadStrData, WriteStrData, FStr<>' ');
end;

procedure TComp_NonePubProps.ReadStrData(Reader: TReader);
begin
    { Reader自动从dfm文件读'StrProp'属性 }
    FStr := Reader.ReadString;
end;

procedure TComp_NonePubProps.WriteStrData(Writer: TWriter);
begin
    { Writer自动将'StrProp'属性写入dfm文件 }
    Writer.WriteString(FStr);
end;

```

TComp_NonePubProps 还演示了如何持久化二进制数据。大家先安装包，然后调用 TComp_NonePubProps.Play 方法，它播放一段 wav，你也可以在放了 TComp_NonePubProps 组件的 Form 上点击右键选择 View as Text 查看持久化的私有数据。

9.7 使用回调函数

回调函数是一个指向一个函数或者过程的指针。通常情况下，我们将一个回调函数 A 的指针变量作为参数传给另一个函数或者过程 B，B 在执行过程中调用 A，这就是所谓的回调。

回调函数大量用在 API 函数中，这些 API 函数常常用来执行一些遍历过程。在遍历过程中，每历到一个结果时，就调用回调函数执行某种处理。

比如一个很重要的 API 函数：

```

BOOL EnumWindows(
    WNDENUMPROC lpEnumFunc,      { 回调函数地址 }
    LPARAM lParam                { 一个自定义变量的地址, 这个自定义变量在
                                lpEnumFunc回调函数中定义 }
);

```

EnumWindows 可用来遍历系统中所有的顶层 (top-level) 窗口。当每遍历到一个顶层窗口时, 它就调用回调函数 lpEnumFunc。

lpEnumFunc 是一个如下类型的函数:

```

BOOL CALLBACK EnumWindowsProc(
    HWND hwnd,                  { 窗口句柄 }
    LPARAM lParam              { 自定义的变量, 可以在EnumWindows中使用 }
);

```

下面就使用这个 API 函数来编一段代码, 这段代码可以遍历出系统当前运行的所有顶层窗口的 Caption。

```

function GetAllRunningWindows: TStrings;
{ 下面的EnumWindowsCode 就是一个回调函数, 注意API函数使用的回调函数必须用
  "stdcall" 修饰 }
function EnumWindowsCode(Wnd: hWnd; Strs: TStrings): Boolean; stdcall;
var
    Buffer: Array[0..MAXBYTE-1] of Char;
begin
    { API函数GetWindowText得到窗口的Caption }
    if GetWindowText(Wnd, Buffer, MAXBYTE) <> 0 then
        Strs.Add(StrPas(Buffer));
    Result := True;
end;
begin
    Result := TStringList.Create;
    EnumWindows(@EnumWindowsCode, LongInt(Result));
    { LongInt(Result) 即传入Result的地址 }
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    with GetAllRunningWindows do
        begin
            ShowMessage(Text);

```



```
Free;
end;
end;
```

其实，我们常常使用的事件的内部过程和回调函数是十分类似的。事件是一个方法类型（大多数情况下是一个过程，不过也可以是函数，但是很少使用。如果要返回值，一般使用 var/out 类型的参数来传递而不是使用函数返回值）的属性。如最常用事件：OnClick 实际上是如下一个类型：

```
TNotifyEvent = procedure (Sender: TObject) of object;
```

我们写的事件代码一般位于类似如下的过程里：

```
procedure TForm1.Button1Click(Sender: TObject);
```

你完全可以将过程 Button1Click 看作是一个回调函数（它在 OnClick 中被回调），只不过它是方法类型的而不是普通过程类型的。

9.8 使用递归算法

通常来说，递归算法的效率比较低，并且耗费空间。但是递归也有其长处，它能使一个蕴含递归关系且结构复杂的程序变得比较简洁精练，相对来说，也增加了可读性。

有些情况下，使用递归算法求解是比较方便的。在本节，我们举两个应用递归算法求解问题的例子。

- (1) 搜索一个文件夹下所有的文件和子文件夹（包括子文件夹的子文件夹）。
- (2) 求 TTreeView 中一个节点的所有子节点的个数（包括子节点的子节点）。

1. 遍历文件夹

在整个例子中，我们要用到 Delphi 定义的两个重要全局函数：

```
function FindFirst(
  const Path: string; Attr: Integer; var F: TSearchRec): Integer;
```

在由 Path 指定的文件夹中找到第一个符合条件的文件或者子文件夹。

参数含义如下：

Path：指定搜索目录。如“c:\MyDir*.*”表示搜索目录“c:\MyDir”下的所有文件和文件夹。

Attr：表示文件属性，可以是以下值之一或者组合：

faReadOnly	只读属性
faHidden	隐藏属性
faSysFile	系统文件属性
faVolumeID	卷标属性

faDirectory 文件夹
faArchive 存档属性
faAnyFile 任何文件和文件夹

F：保存搜索结果。

返回值为 0 时表示搜索到了结果。

```
function FindNext(var F: TSearchRec): Integer;
```

在 FindFirst 或者 FindNext 的基础上继续搜索下一个满足条件的文件和文件夹。

在整个搜索完成时，应该使用过程：

```
procedure FindClose(var F: TSearchRec);
```

释放由 FindFirst 分配的内存。

下面是遍历一个文件夹的源代码：

```
procedure GetDirsAndFiles(
    Path: String; Strings: TStrings; IncludePath: Boolean = True);
{ 参数Path表示起始目录，如"c:\MyDir"；Strings用来存放搜索的结果；IncludePath指
  定返回结果是否包括路径，默认为True}
var
    F: TSearchRec;
    FileName, RFileName: String;
begin
    { 开始搜索}
    if FindFirst(Path+'*.*', faAnyFile, F) = 0 then
    repeat
        FileName := F.Name;
        { "." 和 ".." 两个内建文件夹对我们来说是毫无用处的，但是这个函数也找出来了}
        if (FileName <> '.') and (FileName <> '..') then
        begin
            if IncludePath then
                RFileName := Path + '\' + FileName
            else
                RFileName := FileName;

            { 向字符串列表中添加搜索结果}
            Strings.Add(RFileName);
            if F.Attr and faDirectory <> 0 then      { 如果是文件夹，则递归}
                { 指定新的起始目录}
                GetDirsAndFiles(Path + '\' + F.Name, Strings);
        end;
    until F.Filename = '.';
end;
```

```

until FindNext(F) <> 0;    {直到搜索完成}
FindClose(F);             {最后释放资源}
end;

```

值得一提的是，为了获取一个目录下的文件和文件夹信息，使用上面介绍的递归算法来求解结果并不是惟一的方法。因为获取一个目录下的文件和文件夹信息在 Windows 日常操作中经常用到，比如使用资源管理器，所以 Windows 也定义了一些 API 函数和一些标准消息来实现这类功能。

下面介绍两个与之相关的消息，看如何使用消息来获取目录信息。这两个消息是：

- (1) CB_DIR。它用来将一个文件和文件夹列表加入一个 TComboBox。
- (2) LB_DIR。它用来将一个文件和文件夹列表加入一个 TListBox。

两个消息需要使用的消息参数一样，内部过程也是一样的，只是将得到的结果用不同的控件来显示。

第一个参数 wParam：指定文件夹和文件属性。只有满足这些属性要求的文件夹和文件才会在结果中返回。可以是以下值之一或者它们的组合：

DDL_ARCHIVE	存档属性
DDL_DIRECTORY	目录
DDL_DRIVES	驱动器
DDL_EXCLUSIVE	只包含指定属性的文件和文件夹。比如：
DDL_EXCLUSIVE+DDL_DRIVES：只能取得本机所有驱动器名；	
DDL_EXCLUSIVE+DDL_READONLY：只能取得 lParam 目录下所有只读文件和文件夹名。	
默认情况下包含 DDL_READWRITE，即 DDL_EXCLUSIVE 等价于 DDL_EXCLUSIVE+DDL_READWRITE。	

DDL_HIDDEN	隐藏属性
DDL_READONLY	只读属性
DDL_READWRITE	只包含可读可写属性
DDL_SYSTEM	系统属性

第二个参数 lParam：指定父路径。

以下是一个例子：

```

var
  Path: String;
begin
  {Path为程序运行时所在目录}
  Path := ExtractFilePath(ParamStr(0)) + '*. *';
  {取得所有可读可写文件名列表}
  SendMessage(ComboBox1.Handle, CB_DIR, DDL_READWRITE, Integer(Path));

```

```
SendMessage(ListBox1.Handle, LB_DIR, DDL_READWRITE, Integer(Path));
end;
```

对应的 API 有：

DlgDirList 和 DlgDirListComboBox，将目录信息列入一个 TListBox 和 TComboBox，它们的参数含义和上面两个消息参数是类似的，就不重复解释了。具体可参考 Windows SDK。

2. 求 TTreeNode 的全部子节点个数

我们知道 TTreeNode 有个 Count 属性，但是 Count 返回的是直接子节点（即只是它的子，不包括子的子孙）的个数。如果我们需要知道所有子节点的个数，应该如何做呢？

有两种方法：

（1）使用 TTreeNode 的方法为。

```
function GetNext: TTreeNode;
```

GetNext 返回调用节点相邻的下一个节点，即按从上到下的顺序，而不关心父子关系。我们只需要一直 GetNext，到不再存在节点或者节点的层次（即 Level 属性）变小为止。

下面是我编写的源代码：

```
function GetAllChildNodeCount(Node: TTreeNode): Integer;
var
  CurrNode: TTreeNode;
  Level: Integer;
begin
  Result := 0;
  CurrNode := Node;
  Level := CurrNode.Level;

  CurrNode := CurrNode.GetNext;
  { 不停地取得当前节点的下一个节点，直到下一个节点不存在或者层次不符为止 }
  while (CurrNode <> nil) and (CurrNode.Level > Level) do
  begin
    Inc(Result);
    CurrNode := CurrNode.GetNext;
  end;
end;
```

（2）利用 Count 属性结合递归算法求解。

```
function GetAllChildNodeCount2(Node: TTreeNode): Integer;
var
  I, ACount: Integer;
```



```

CurrNode: TTreeNode;
begin
  ACount := Node.Count;
  for I := 0 to Node.Count-1 do
  begin
    CurrNode := Node[I];
    if CurrNode.HasChildren then
      { 递归求解 }
      Inc(ACount, GetAllChildNodeCount2(CurrNode));
    end;
  Result := ACount;
end;

```

9.9 编写 NT 服务程序

NT 服务程序被大量使用，如 Web 服务程序、数据库服务程序等。安装完成 Oracle 数据库服务器后，就会注册好几个以“Oracle”开头命名的服务。服务程序在系统启动后（即使没有用户登录）就开始运行了，所以它可以完成很多后台工作。

在 Delphi 的 SvcMgr 单元，提供两个重要的类：

TSERVICEApplication 和 TService。

利用这两个类，可以非常方便地编写 NT 服务程序。一个 NT 服务程序，由一个 TServiceApplication 实例和一个或者多个 TService 实例构成。TServiceApplication 提供一个服务程序框架，而每个 TService 则代表一个具体的服务。一个服务程序中可以包含多个服务，对应的就有多个 TService。和一个普通应用程序相比，TServiceApplication 相当于 TApplication，TService 相当于 TDataModule 和 TForm（TService 实际就是 TDataModule 的派生类）。服务程序和普通程序一样都有一个全局变量 Application（定义在 SvcMgr 单元），但是服务程序的 Application 是 TServiceApplication 类型（所以一个单元中不要同时包含 Forms 和 SvcMgr 单元，以免发生混乱）。

当注册完成的服务程序时，其中包含的所有服务都会被注册，打开“控制面板 | 管理工具 | 服务”（即服务控制管理器，Service Control Manager，SCM），我们看到该 TServiceApplication 中所有的 TService。

Delphi 提供方便的服务程序编写向导。选择菜单 File|New|Other|Service Application，可以生成一个服务程序框架，并自动添加一个服务。继续在同样的界面中选择 Service 可以添加多个服务。这时候按 F9 键，服务程序就运行了。当然这时候这个服务什么也不能作，也没有注册到 SCM 中，它只是在那里莫名其妙地运行而已。

因为 TService 是 TDataModule 的派生类，所以它在运行时是不可见的，设计程序，我们也只能将不可视(nonVisible)组件放到它上面。如果一个服务程序需要可视化界面，则可以向其中添加 TForm。

添加的第一个 TForm 就是服务程序的主界面。

以下是一些服务控制命令：

(1) 安装服务到 SCM：

服务程序名（须包含全部路径） /install

(2) 从 SCM 卸载服务：

服务程序名（须包含全部路径） /uninstall

如果在以上两个命令后再加上参数：/silent，那么安装和卸载后不会有提示信息弹出。

(3) 启动服务：

net start 服务名

(4) 暂停服务：

net pause 服务名

(5) 停止服务：

net stop 服务名

上述命令也可以用程序来完成，如：

```
var
    SvrName, CommandLine: String;
begin
    SvrName := 'FilesDownloadSRV';
    CommandLine := 'net start ' + SvrName;
    WinExec(PChar(CommandLine), SW_HIDE);
    {WinExec是16位系统运行命令行的一个API函数，Win32环境应该使用CreateProcess代
      替。这里为了简化，直接使用了WinExec。参数SW_HIDE表示不弹出命令行运行环境——
      否则它黑漆漆的面孔一闪而过，怪吓人的☺}
end;
```

9.10 编写只能惟一运行的程序

在 CSDN 上常常看到有人询问如何编写只能惟一运行的程序。惟一运行是指不能在一个机器上同时运行相同程序的多个实例。

这种程序功能的实现，本质上是在程序加载时到系统中去搜索是否已经有同样程序的实例在运行。如果有，就不要加载了，退出去。

关键的是“搜索”。怎么搜索呢？我们向某个单位找人，常常是问有没有一个名字叫张三的人存在。所以，应按惟一特征来搜索，并且必须事先知道这个惟一特征。

很多朋友把应用程序主窗体的标题（Caption，如“Form1”）或者类名（如“TForm1”）等来作为这个惟一特征。比如我们选择 File|New|Application，新建一个最简单的应用程序，然后选择菜单



Project|View Source，写如下代码：

```
program Project1;

uses
  Windows,
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  if FindWindow('TForm1', 'Form1') = 0 then
    { 当再运行Project1.exe时，找到已经运行的主窗体Form1的句柄不等于0，所以直接退出程
      序。想想的确是这样，还挺简单的！ }
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end;
end.
```



然后按 F9 键……嗯，为什么不能启动程序？很明显，此时 FindWindow('TForm1', Form1) <> 0，奇怪，不是还没有使用 CreateForm 创建 Form1 吗？

别忘了，Delphi 也是一个应用程序，这时候它里面就包含一个窗体 TForm1，而且你正在设计这个窗体！所以 FindWindow 不等于 0。因此，还得选择 File|Close All 关掉 Form1，这时候到资源管理器中运行 Project1.exe，双击运行之，出来了；再双击，不能运行了。

可见这种方法可行，但是很不可靠。要知道两个不同程序的主窗体标题和类名相同是很正常的事情。如果用上面的方法，很可能运行 Project1.exe 后就无法运行 Project2.exe。

因此，应该选择一个真正惟一的特征。我们可以生成这样一个特征。大家可能知道 Windows 系统有很多内核对象。内核对象可以使用命名来创建，且名字和内核对象一一对应。很显然，如果用 GUID 号来命名内核，那么就能获得惟一内核对象，因为每次生成的 GUID 都是不同的。

下面举个例子，在这个例子中，我们选择互斥（Mutex）内核。

选择菜单 File|New|Application 新建一个工程，再选择 File|New|Unit 新建一个单元 Anti_MoreApps，在该单元写入如下代码：

```
unit Anti_MoreApps;
```

```

interface

    function HaveApp(AppID: String): Boolean;
    { AppID是一个GUID号,函数搜索以AppID命名的互斥内核,如找到即表示已经有一个相同的
      应用程序运行并创建了这个内核。 }

implementation

uses Windows;

var
    HMutex: THandle;

function HaveApp(AppID: String): Boolean;
begin
    Result := False;
    HMutex := OpenMutex(MUTEX_ALL_ACCESS, False, PChar(AppID));
    { 搜索内核 }
    if HMutex = 0 then      { 没找到那么创建该命名的内核 }
        HMutex := CreateMutex(nil, False, PChar(AppID))
    else Result := True;
end;

initialization

finalization
    if HMutex <> 0 then CloseHandle(HMutex); { 创建的内核在不需要使用时应该销毁 }

end.

```

选择Project|View Source, 写入如下代码:

```

.....

begin
    if not HaveApp('{51991952-44C3-4B25-A936-9FD9E05B53AA}') then
    { 这个GUID号可通过快捷键Ctrl+Shift+G生成 }
    begin
        Application.Initialize;
        Application.CreateForm(TForm1, Form1);
        Application.Run;
    end;
end;

```




```
end;  
end.
```

现在大家运行 Project1.exe，是不是很满意？

9.11 字段类型全家福

Delphi 中定义了 31 个字段类型用来表示所有的数据库字段类型。这些字段的关系如下所示：

```
*TNumericField  
    TIntegerField  
        TSmallintField TAutoIncField TWordField  
    TLargeintField  
    TFloatField  
        TCurrencyField  
    TBCDField  
    TFMTBCDField  
TStringField  
    TWideStringField TGuidField  
TDateTimeField  
    TDateField TTimeField  
TBlobField  
    TMemoField TGraphicField  
TBooleanField  
TVariantField  
TAggregateField  
TSQLTimeStampField  
TBinaryField  
    TBytesField  
    TVarBytesField  
*TObjectField  
    TADTField TReferenceField TArrayField TDataSetField  
TInterfaceField  
TIDispatchField
```

其中加*的 TNumericField 和 TObjectField 分别是数字和对象类型字段的基类，不作为具体的字段类型使用，所以真正直接使用的字段共 31 个。当然，所有字段类的祖先类都是 TField。

TField 有个 DataType (TFieldType 类型) 属性，它用一些常数来标识字段的类型，基本上和上面的字段类型树是一一对应的。



下面，我对这些字段作个扼要介绍。

1. TNumericField

数据类型字段的基本类型。

TIntegerField：普通整数类型字段；TSmallIntField：小整数类型字段；TAutoIncField：自动增加整数类型字段；TWordField：Word 整数类型字段。

TLargeIntField：大整数类型字段。

TFloatField：浮点类型字段。TCurrencyField：货币类型字段。

TBCDField 和 TFMTBCDField：BCD（二进制编码的十进制树）类型字段。常常用来表示货币，TFMTBCDField 比 TBCDField 能达到的精度更高，但是运行效率要低一些。

2. TStringField

字符串类型字段。

TWideStringField：宽字符串类型字段。

TGuidField：GUID（全球惟一标识符）类型字段。

3. TDateTimeField

日期时间类型字段。

TDateField：日期类型字段。

TTimeField：时间类型字段。

4. TBlobField

BLOB（二进制大型对象）类型字段。

TMemoField：Memo 类型字段。一般地，如果在数据库中定义的字符串字段的长度超过 255 时，使用 Delphi 连接这个字段时，字段类型为 TMemoField。

TGraphicField：图形图像类型字段。

5. TBooleanField

布尔类型字段。有些数据库不能直接定义布尔类型字段，而用 Char(1)、VarChar(1)或者整数字段等代替。

6. TVariantField

可变数据类型字段。

7. TAggregateField

统计字段。一般用在客户数据集中，实现对其他字段数据的统计功能，大多数数据库中不能直接定义这种字段类型。

8. TSQLTimeStampField

用在 DBExpress 数据集中的日期时间类型字段。

9. TBinaryField

二进制数据类型字段。通常直接定义二进制数据类型字段来管理无类型的数据。



TBytesField：字节类型字段，其字节个数是固定的。TVarBytesField：可变字节数类型字段，其字节个数是可变的。它们类似于 Char 和 VarChar 字段的关系。

10. TObjectField

对象类型字段的基类。

TADTField：ADT（抽象数据类型）类型字段。用来管理类似记录形式的复杂数据。

TReferenceField：对象引用类型字段。通过指针或者引用来管理对象。

TArrayField：数组类型字段。

TDataSetField：数据集类型字段。用来存取网状（nested）数据集数据。

11. TInterfaceField

接口类型字段。

TIDispatchField：派遣接口类型字段。

9.12 获取数据库结构信息

当我们使用 Delphi 的数据库连接组件连接到数据库后，常常需要通过程序获取数据库的一些结构信息。

比如在一个数据库备份程序中，需要提供一个数据库中表的清单让用户选择，然后只备份被选择的表。

我们不能事先将所有表的名字装入一个 TListBox，而应该在程序运行时从数据库服务器获取，否则就成了死的东西，没有变化的余地。

幸运的是，Delphi 的一些数据库组件提供了一些方法，这些方法可以让我们在运行时获取数据库的结构信息。这类方法有如下一些：

1. TDatabase 类

(1) 获取所有表名：

```
procedure GetTableNames(List: TStrings; SystemTables: Boolean = False);
```

返回的所有表名存放在 List 中。SystemTables 指定是否也返回系统表（即数据库为了内部操作而自动创建的）的名字。

(2) 获取表的字段名：

```
procedure GetFieldNames(const TableName: String; List: TStrings);
```

List 返回表 TableName 的所有字段名。

2. TSession 类

一般地，一个 TDatabase 对象会自动建立一个 TSession 对象。因此，可以通过属性 TDatabase.Session 来获得 Session 对象。

(1) 获取数据库名列表：



```
procedure GetDatabaseNames(List: TStrings);
```

(2) 获取数据库别名列表：

```
procedure GetAliasNames(List: TStrings);
```

(3) 获取数据库别名对应的驱动程序名：

```
function GetAliasDriverName(const AliasName: String): String;
```

(4) 获取驱动程序名列表：

```
procedure GetDriverNames(List: TStrings);
```

(5) 获取表名列表：

```
procedure GetTableNames(const DatabaseName, Pattern: String; Extensions,  
    SystemTables: Boolean; List: TStrings);
```

(6) 获取字段名列表：

```
procedure GetFieldNames(const DatabaseName, TableName: string;  
    List: TStrings);
```

(7) 获取存储过程名列表：

```
procedure GetStoredProcNames(const DatabaseName: String; List: TStrings);
```

3. TADOConnection 类

(1) 获取表名列表：

```
procedure GetTableNames(List: TStrings; SystemTables: Boolean = False);
```

(2) 获取字段名列表：

```
procedure GetFieldNames(const TableName: String; List: TStrings);
```

(3) 获取存储过程名列表：

```
procedure GetProcedureNames(List: TStrings);
```

这些方法的用法都很简单，就不一一举例了。

9.13 深入使用 TCanvas

TCanvas 是 VCL 中非常重要的类，直接派生于 TObject。它提供给那些需要自己画表面的控件使用，如 TImage、TCustomForm、TPrinter、TGraphicControl 等，在 Delphi 在线帮助的索引中键入“Canvas”可以看到所有使用 TCanvas 的类。而 TEdit、TListbox 等 Windows 标准控件不需要 TCanvas 支持，系统会画出它们。

TCanvas 主要提供以下功能：

- (1) 可以指定刷子、画笔和字体，以实现不同绘画效果。
- (2) 可以填充和绘画图形。
- (3) 可以输出文字。



- (4) 可以绘出图像。
- (5) 可以响应图形图像变化。

TCanvas 的重要属性：

property Brush: TBrush;

画刷。用于填充，可以设置填充的图像、图案、颜色和填充方式。

在 TBrush 的子属性中重点讲 Style。Style 表示画刷是用什么模式进行绘制的（见表 9-1）。

property Style: TBrushStyle;

表 9-1

值	含 义
bsSolid	单色
bsClear	透明
bsBDiagonal	45 度斜线
bsFDiagonal	-45 度斜线
bsCross	横、竖线交叉
bsDiagCross	45、-45 度斜线交叉
bsHorizontal	横线
bsVertical	竖线

如果要自定义，可以用 TBrush.BitMap 属性，设置 8×8 像素位图。

property Font: TFont;

字体。设置输出文字的字体。

property Handle: HDC;

画布所依附的图形设备接口的句柄。

property Pixels[X, Y: Integer]: TColor;

得到指定像素处的颜色。

property ClipRect: TRect;

一个矩形区域，所有绘制只能在这个矩形区域进行。它是只读的。

property Pen: TPen;

画笔。设置画出的直线、矩形、椭圆等的线型、粗细、颜色、宽度等。

在 TPen 的子属性中重点讲 Mode。Mode 表示 TPen.Color 和 Canvas 背景色是如何相互作用的（见表 9-2）。

property Mode: TPenMode;



表 9-2

分类	Mode	作用色		结果像素颜色
单色	pmBlack			总是黑色
	pmWhite			总是白色
	pmNop			画布背景色
	pmCopy			TPen.Color
混色 1	pmMergePenNot	TPen.Color	Canvas 背景色的反色	两种作用色作或运算(OR)
	pmMergeNotPen	TPen.Color 反色	Canvas 背景色	
	pmMerge	TPen.Color	Canvas 背景色	
混色 2	pmMaskPenNot	TPen.Color	Canvas 背景色的反色	两种作用色作与运算(AND)
	pmMaskNotPen	TPen.Color 反色	Canvas 背景色	
	pmMask	TPen.Color	Canvas 背景色	
混色 3	pmXor	TPen.Color	Canvas 背景色	两种作用色作异或运算(XOR)
反色	pmNot	Canvas 背景色		作用色的反色 (NOT)
	pmNotCopy	TPen.Color		
	pmNotMerge	pmMerge		
	pmNotMask	pmMask		
	pmNotXor	pmXor		

property CopyMode: TCopyMode **default** cmSrcCopy;

它和 TPen.Mode 类似，只不过是设定 TCanvas.CopyRect 的作用方式（见表 9-3）。

表 9-3

CopyMode	含 义
cmBlackness	将区域涂成黑色
cmWhiteness	将区域涂成白色
cmDstInvert	将区域图像翻转(NOT)(这时候 CopyRect 中的源位图不起作用)
cmMergeCopy	将区域图像和源位图作 AND 运算
cmMergePaint	将区域图像和翻转后的源位图作 OR 运算
cmNotSrcCopy	复制翻转后的源位图
cmNotSrcErase	将区域图像和源位图作 OR 运算，然后翻转



续表

CopyMode	含 义
cmPatCopy	复制源模式
cmPatInvert	将区域图像和源模式作 XOR 运算
cmPatPaint	将翻转后的源位图和源模式作 OR 运算；再将结果和区域图像作 OR 运算
cmSrcAnd	和 cmMergeCopy 相同
cmSrcCopy	复制源位图。默认
cmSrcErase	将翻转后的区域图像和源位图作 AND 运算
cmSrcInvert	将区域图像和源位图作 XOR 运算
cmSrcPaint	将区域图像和源位图作 OR 运算

TCanvas 的重要方法：

procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);

等，用来绘制弧线、直线、矩形等。

procedure BrushCopy(const Dest: TRect; Bitmap: TBitmap;
 const Source: TRect; Color: TColor);

从位图复制图像到画布。

procedure CopyRect(
 const Dest: TRect; Canvas: TCanvas; const Source: TRect);

从别的画布复制图像到画布。

procedure Draw(X, Y: Integer; Graphic: TGraphic);

绘制图像。

procedure StretchDraw(const Rect: TRect; Graphic: TGraphic);

拉伸绘制图像。

procedure FillRect(const Rect: TRect);

填充指定区域。

procedure FloodFill(X, Y: Integer; Color: TColor; FillStyle: TFillStyle);

另一种填充方法。

procedure DrawFocusRect(const Rect: TRect);

绘制焦点区域。

procedure FrameRect(const Rect: TRect);

绘制区域边框。

procedure MoveTo(X, Y: Integer);

移动画笔位置。



```
procedure TextOut(X, Y: Integer; const Text: string);
```

输出文字。

```
function TextExtent(const Text: string): TSize;
```

在当前字体设置下，一个字符串输出到画布上需要占用的高度和宽度。

```
function TryLock: Boolean;
```

锁定画布，不允许别的线程绘制它。

```
procedure Unlock;
```

解锁画布。

TCanvas 的重要事件：

```
property OnChange: TNotifyEvent;
```

画布图像改变后触发。

```
property OnChanging: TNotifyEvent;
```

画布图像即将改变时触发。

TCanvas 的子类：

TControlCanvas

图形控件使用，如 TImage、TSpeedButton 等。

TMetafileCanvas

用于绘制 WMF 类型图像。

下面看几个例子：

1. 当鼠标移动时，标出鼠标所在窗口控件的范围

```
var
    ACanvas: TCanvas;

procedure TForm1.FormCreate(Sender: TObject);
begin
    ACanvas := TCanvas.Create;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    ACanvas.Free;
end;
```




```

procedure TForm1.Timer1Timer(Sender: TObject);
var
    CurrPoint: TPoint;
    FindHandle: HWND;
    H: HDC;
    FindRect: TRect;
begin
    CurrPoint := ScreenToClient(Mouse.CursorPos);
    {Mouse 是VCL 定义的全局变量, Mouse.CursorPos 的坐标系是屏幕, 所以使用
      Form1.ScreenToClient 将坐标系转化到Form1上。}
    FindHandle := ChildWindowFromPoint(Handle, CurrPoint);
    {调用API 在Form1上指定点处找子窗口句柄}
    if FindHandle <> 0 then
    begin
        Windows.GetClientRect(FindHandle, FindRect);           {找到子窗口外框}
        ACanvas.Lock;      {锁定画布, 防止其他线程操作画布}
        H := GetDCEx(FindHandle, 0, DCX_CACHE or DCX_CLIPSIBLINGS);
        {调用API 得到子窗口的GDI 句柄}
        SetViewportOrgEx(H, FindRect.Left, FindRect.Top, nil);
        {将视角转换到子窗口的左上角}
        ACanvas.Handle := H;
        {设置画布句柄为子窗体GDI 句柄, 没有句柄, 将无法绘制图形图像。}
        ACanvas.DrawFocusRect(Rect(0, 0, FindRect.Right, FindRect.Bottom));
        {画外框。DrawFocusRect 使用XOR画法, 所以第二次画会清掉第一次结果, 这样就可以出
          现闪烁效果。}
        ACanvas.Unlock;
    end;
end;

```

上面的例子融合了画布的句柄使用、图形绘制方法、坐标转换、如何通过父窗口找子窗口、如何得到控件的 GDI 句柄等许多技巧。根据这个例子提供的思路, 可以做一个 Microsoft HTML Help Image Editor 那样的窗口捕捉工具。但是它是在后台运行, 捕捉当前窗口的子窗口, 所以需要作很多改动:

- (1) 去掉定时器, 建立一个鼠标移动的钩子函数, 通过回调函数实现鼠标移动事件的反馈。
 - (2) 在回调函数中调用 API 之 GetForegroundWindow 找到当前父窗口, 并根据鼠标位置 (在回调函数参数里) 找到子窗口。
 - (3) 接下来就基本相同了。
2. 如何把某个控件在窗体上显示的图像保存到文件

```

procedure TForm1.SaveToFile;
var
    sourceRect, destRect: TRect;
begin
    with ACanvas do
    begin
        destRect := Rect(0, 0, ClipRect.Right - ClipRect.Left,
            ClipRect.Bottom - ClipRect.Top); {取得ACanvas的整个大小}
        sourceRect := destRect;
        Image1.Canvas.CopyRect(destRect, ACanvas, sourceRect);
        {将ACanvas上的图像(即控件图像)全部复制到Image1}
        Image1.Picture.SaveToFile('F:\aa.bmp'); {保存到文件}
    end;
end;

```

3. 如何制作 WMF 文件以及如何显示该文件的图像

WMF (Windows MetaFile) 即 Windows 元文件。它不像位图 (BMP) 那样保存实际图像的像素信息, 所以本质上并不是图像文件。它保存图像如何画出来的信息, 也就是一系列 GDI 函数。比如通过几个圆、多少点构成一个烧饼图像。元文件分为两种: 16 位 (.wmf) 和 32 位 (.emf , Enhanced MetaFile)。

下面看如何制作 WMF 文件:

```

procedure TForm1.Button1Click(Sender: TObject);
var
    Mfile: TMetaFile;
    MCanvas: TMetaFileCanvas;
    R: TRect;
begin
    Mfile := TMetaFile.Create;
    MCanvas := TMetaFileCanvas.Create(Mfile, 0);
    {将TMetaFileCanvas和TMetaFile关联}
    {以下的所有绘制实际上都画到了Mfile上。}
    with MCanvas do
    begin
        Brush.Color := clRed;
        Ellipse(0,0,100,100);
        R := Rect(Button1.Left, Button1.Top, Button1.Left + 100,
            Button1.Top + 30);
        CopyRect(Rect(0, 0, 100, 50), Canvas, R);
        Free;
    end;
end;

```

```
{ 必须首先销毁MCanvas ,因为它占用了Mfile。下面这句将一个元文件的图像显示到Form上 ;  
紧接着一句用于保存图像到文件(从文件读取用LoadFromFile)。}  
Form1.Canvas.Draw(0,0,Mfile);  
Mfile.SaveToFile('H:\AA.emf');  
Mfile.Free;  
end;
```

9.14 指针列表类的使用

指针列表类,用来管理一系列指针。这些指针可以是指向对象、接口、类引用等既有数据类型以及其他任何自定义的数据类型。

Delphi 中常用的指针列表类有以下一些:

1. 无序类

这类列表的项与顺序无关,即是说某个指针在列表的第几项是无关紧要的。在这类列表中查找项时,常常是使用遍历方法。

(1) TList。管理任何类型指针列表。它提供了最大的灵活性,但是使用起来也最麻烦。因此,在实现特定功能时,一般都使用它的子类或者其他列表类。

(2) TObjectList。对象指针列表。它派生于 TList,增加了管理 Object 内存的能力。即当属性 OwnsObjects 为 True 时(默认值),删除某列表项或者销毁整个列表时,列表项对应的对象被自动销毁。

(3) TComponentList。组件指针列表。它派生于 TObjectList,增加了自动跟踪 Component 的能力:当一个 Component 被销毁时,TComponentList 的对应项会被自动删除。

(4) TClassList。类引用列表。它派生于 TList,专门管理类引用。

(5) TInterfaceList。接口列表。

2. 有序类

这类列表中各项的顺序是至关重要的。如果顺序乱了,列表就不能发挥作用。对于这类列表中项的存取,总是从头或者尾进行,而不能从中间某个位置直接插入。

(1) TStack。模拟栈,它的项总是后进先出的。

(2) TObjectStack。也是模拟栈,不过它的项是对象指针。

(3) TQueue。模拟队列,它的项总是先进先出的。

(4) TObjectQueue。也是模拟队列,不过它的项是对象指针。

我们看下面的一个例子:

```
unit Unit1;  
  
interface
```

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls, Contnrs;

type

TForm1 = **class**(TForm)

Button1: TButton;

Button2: TButton;

Button3: TButton;

Button4: TButton;

ListBox1: TListBox;

ListBox2: TListBox;

ListBox3: TListBox;

ListBox4: TListBox;

Label1: TLabel;

Label2: TLabel;

Label3: TLabel;

Label4: TLabel;

Button5: TButton;

Button6: TButton;

Label5: TLabel;

procedure FormCreate(Sender: TObject);

procedure FormDestroy(Sender: TObject);

procedure Button5Click(Sender: TObject);

procedure Button1Click(Sender: TObject);

procedure Button2Click(Sender: TObject);

procedure Button3Click(Sender: TObject);

procedure Button4Click(Sender: TObject);

procedure Button6Click(Sender: TObject);

private

ObjectList: TObjectList;

ComponentList: TComponentList;

ObjectStack: TObjectStack;

ObjectQueue: TObjectQueue;

public

{ Public declarations }

end;

var

Form1: TForm1;





```
implementation
```

```
{ $R *.dfm }
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
    ObjectList := TObjectList.Create(False);
```

```
    ComponentList := TComponentList.Create(False);
```

```
    ObjectStack := TObjectStack.Create;
```

```
    ObjectQueue := TObjectQueue.Create;
```

```
end;
```

```
procedure TForm1.FormDestroy(Sender: TObject);
```

```
begin
```

```
    ObjectList.Free;
```

```
    ComponentList.Free;
```

```
    ObjectStack.Free;
```

```
    ObjectQueue.Free;
```

```
end;
```

```
{ 在Button5单击事件中初始化所有列表，将Form1的所有子控件加入各列表 }
```

```
procedure TForm1.Button5Click(Sender: TObject);
```

```
var
```

```
    C: Integer;
```

```
begin
```

```
    for C := 0 to ControlCount-1 do
```

```
    begin
```

```
        ObjectList.Add(Controls[C]);
```

```
        ComponentList.Add(Controls[C]);
```

```
        ObjectStack.Push(Controls[C]);
```

```
        ObjectQueue.Push(Controls[C]);
```

```
    end;
```

```
end;
```

```
{ 点击Button5 初始化各列表后点击Button6 销毁Button5。我们要看看各列表如何处理指向的  
    对象已经被销毁的列表项 }
```

```
procedure TForm1.Button6Click(Sender: TObject);
```

```
begin
```

```
    Button5.Free;
```

```
end;
```

```
{ 将ObjectList所有控件的名字列入ListBox1}  
procedure TForm1.Button1Click(Sender: TObject);  
var  
    C: Integer;  
begin  
    ListBox1.Items.Clear;  
    for C := 0 to ObjectList.Count-1 do  
        ListBox1.Items.Add(TControl(ObjectList.Items[C]).Name);  
end;  
  
{ 将ComponentList所有控件的名字列入ListBox2}  
procedure TForm1.Button2Click(Sender: TObject);  
var  
    C: Integer;  
begin  
    ListBox2.Items.Clear;  
    for C := 0 to ComponentList.Count-1 do  
        ListBox2.Items.Add(ComponentList.Items[C].Name);  
end;  
  
{ 将ObjectStack所有控件的名字列入ListBox3}  
procedure TForm1.Button3Click(Sender: TObject);  
var  
    C: Integer;  
begin  
    ListBox3.Items.Clear;  
    for C := 0 to ObjectStack.Count-1 do  
        ListBox3.Items.Add(TControl(ObjectStack.Pop).Name);  
end;  
  
{ 将ObjectQueue所有控件的名字列入ListBox4}  
procedure TForm1.Button4Click(Sender: TObject);  
var  
    C: Integer;  
begin  
    ListBox4.Items.Clear;  
    for C := 0 to ObjectQueue.Count-1 do  
        ListBox4.Items.Add(TControl(ObjectQueue.Pop).Name);  
end;  
  
end.
```



上面程序的运行情况如图 9-2 所示。运行后首先点击 Button5 (初始化各列表), 然后点击 Button6 (销毁 Button5), 最后点击 Button1...Button4 (取出各列表的各项对象的 Name 属性)。

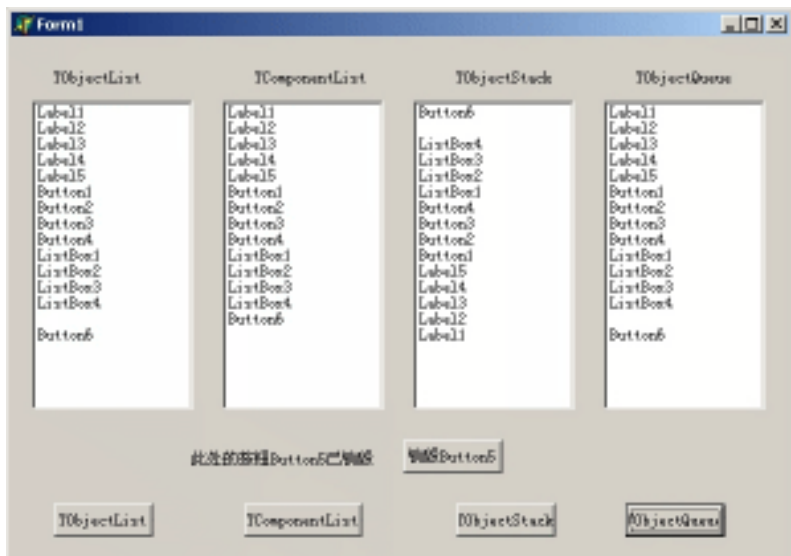


图 9-2 使用指针列表类

从图 9-2 可以看到：

(1) 从各类列表取出项时的顺序。

(2) 除了 TComponentList 外, 被销毁项 Button5 对应位置显示的 Name 属性值为空, 因为 TComponentList 具有自动跟踪能力。当 Button5 被销毁后, TComponentList 已经自动删除了 Button5 在列表中的对应项。

9.15 结构化存储技术

Word 或者 WPS 文档大家经常使用, 都很熟悉了。那么我们有没有考虑过它的文件是怎么保存的? 拿一个 doc 文件来说, 它里面包含文字、图片甚至声音、视频等, 其中的多媒体数据可能是从外部文件插入的, 但是最终都只是保存了一个单独的 doc 文件, 而不是将各种多媒体内容分离出来保存为单独的附属文件。

你也许认为它是使用了流的技术, 就是将所有文件的数据按照一定的顺序写进一个流中, 最后保存为一个 doc 文件。这是一种很好的想法, 很多软件最终保存的文件都是使用这种方式生成的, 但是 doc 文件却并非使用简单的流来保存, 而是使用了结构化存储技术。不过, 要理解结构化存储, 则必须首先理解流保存, 因为结构化存储的基础就是流技术。所以, 在讲解结构化存储技术之前, 还是先看看如何用单独的一个流来存取多个文件。

在光盘对应目录下，有三个文件：乖小孩.gif、浴血长沙——第三次长沙会战守城记.htm、双胞胎.jpg。我们来考虑如何将这三个文件保存为一个单独的文件，然后再从这个单独的文件中读出原始的三个文件。

要解决这个文件，必须要使用流分域技术。打个比方，我们的单身公寓共六层，指定第 5 楼住女孩，第 1 楼住部门 A 的男孩，第 2 楼住部门 B 的男孩……那么要找女孩时，就直接到第 5 楼，找部门 B 的男孩，直接到第 2 楼。

流分域也是这样，将整个流分成多段，每段存储一个子文件。因为每个子文件的大小是不定的，所以不可能固定分配每段的大小。否则，段小了则无法容纳子文件的内容，大了则造成浪费。因此，每段还必须分为一些小段，小段中要保存段的大小（即子文件长度）。

这样可以构造流的内部结果如图 9-3 所示。

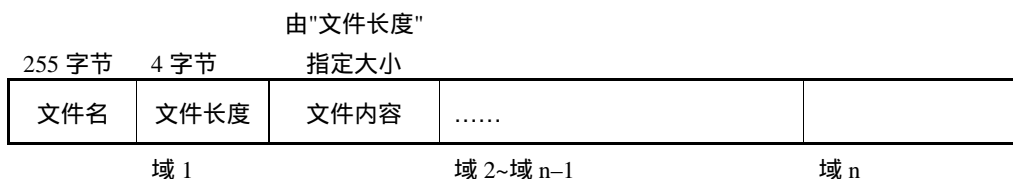


图 9-3 流分域示意图

拿域 1 来说，它保存了子文件 1 的所有信息。其子域文件名被固定分配 255 字节，用来保存子文件的名字；文件长度被固定分配 4 字节，用来保存子文件的长度；子域文件内容用来保存子文件的实际数据，这个子域的长度保存在子域文件长度中。

因此，读取某个子文件时，只需要首先读出 255 字节的文件名，再读取 4 字节的文件长度，最后读出文件长度字节的文件实际内容即可。

下面一段代码演示了如何使用这种流分域技术来存取多个文件。为了编码方便，我们在整个流的头部又分出了 4 字节来保存子文件的个数。

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  {记录TFileInfo描述了流的一个域的结构}
  TFileInfo = packed record
    FileName: String[255];
```



```

    FileSize: Integer;
    FileData: TMemoryStream;
end;
{ 动态数组TFilesInfo描述了整个流的结构 }
TFilesInfo = Array of TFileInfo;

TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
private
    Path: String;
    { 将多个子文件转化为一个流 }
    procedure FileInfoToStream(FileInfo: TFilesInfo; Stream: TStream);
    { 将整个流转化为多个子文件 }
    function StreamToFileInfo(Stream: TStream): TFilesInfo;
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
    I: Integer;
    FileInfo: TFilesInfo;
    SaveStream: TMemoryStream;
begin
    SetLength(FileInfo, 3);
    { 初始化动态数组FilesArray }
    for I := Low(FileInfo) to High(FileInfo) do
        with FileInfo[I] do
            begin
                case I of

```

```

0: FileName := '乖小孩.gif';
1: FileName := '浴血长沙 ——第三次长沙会战守城记.htm';
2: FileName := '双胞胎.jpg';
end;
FileData := TMemoryStream.Create;
FileData.LoadFromFile(Path+FileName);
FileSize := FileData.Size;
end;

SaveStream := TMemoryStream.Create;
{ 将子文件转化为一个流SaveStream }
FilesInfoToStream(FilesInfo, SaveStream);
{ 将流SaveStream保存为一个文件, 这样, 多个子文件就被合成了一个大文件 }
SaveStream.SaveToFile(Path+'lxpbuaa.data');

for I := Low(FilesInfo) to High(FilesInfo) do
  FilesInfo[I].FileData.Free;
SaveStream.Free;
end;

procedure TForm1.FilesInfoToStream(
  FilesInfo: TFilesInfo; Stream: TStream);
var
  I: Integer;
begin
  I := Length(FilesInfo);
  { 首先在流的头4个字节写入域个数 (即子文件的个数) }
  Stream.Write(I, 4);
  { 将所有子文件信息写入流 }
  for I := Low(FilesInfo) to High(FilesInfo) do
    with FilesInfo[I], Stream do
      begin
        { 注意读写字符串时用FileName[1]而不是FileName, 否则它从0位置开始, 这样就不对了 }
        Write(FileName[1], Sizeof(FileName)-1);
        Write(FileSize, SizeOf(FileSize));
        CopyFrom(FileData, FileData.Size);
      end;
    Stream.Position := 0;
  end;
end;

```



```
procedure TForm1.Button2Click(Sender: TObject);
var
  I: Integer;
  FilesInfo: TFilesInfo;
  LoadStream: TMemoryStream;
begin
  LoadStream := TMemoryStream.Create;
  { 将整个大文件加载到流 }
  LoadStream.LoadFromFile(Path+'lxpbuaa.data');
  { 分离流 }
  FilesInfo := StreamToFilesInfo(LoadStream);
  LoadStream.Free;

  { 将分离出的流还原为文件 }
  for I := Low(FilesInfo) to High(FilesInfo) do
    with FilesInfo[I] do
      begin
        FileData.SaveToFile(Path + 'Copy_' + Trim(FileName));
        FileData.Free;
      end;
    end;

function TForm1.StreamToFilesInfo(Stream: TStream): TFilesInfo;
var
  I: Integer;
  Buffer: String[255];
begin
  Stream.Position := 0;
  { 首先读取域个数 (即子文件个数) }
  Stream.Read(I, 4);
  SetLength(Result, I);
  for I := Low(Result) to High(Result) do
    with Result[I], Stream do
      begin
        { 读取文件名 }
        Read(Buffer[1], SizeOf(Buffer)-1);
        FileName := Buffer;
        { 读取文件长度 }
        Read(FileSize, 4);
        FileData := TMemoryStream.Create;
        { 读取文件长度字节的文件内容 }
```

```

    FileData.CopyFrom(Stream, FileSize);
    Seek(soFromCurrent, FileSize);
end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    Path := ExtractFilePath(ParamStr(0));
end;

end.

```

运行上面的代码,先按 Button1 合成文件,再按 Button2 分离文件。最后我们发现,的确生成了三个文件:Copy_乖小孩.gif、Copy_双胞胎.jpg、Copy_浴血长沙——第三次长沙会战守城记.htm,且都没有数据损坏。

大部分软件保存不是很复杂的结果文件时,都是采用上面所讲的流分域技术。比如游戏《三国志 9》的新武将登录的结果文件。在光盘上附带了我写的一个小工具“pLgnPrsn.exe”,该工具用来给游戏《三国志 9》登录新武将。玩过《三国志 9》的朋友都知道,使用游戏自带的登录功能无法输入简体中文,因此,登录的武将的名字都是乱码。我的这个工具解决了这个问题,并将武将的大部分属性公开出来,这样可以十分容易地登录超级武将。这个工具也可以从一些游戏网站上下载,如:

http://www.9.pconline.com.cn/pcgames/gamelist.php?kind_id=1002&taxis=2&pn=-1。

流分域存储也可以用在数据库开发中,将合成的整个流存入一个 blob 字段。

但是流分域存储有一个很大的问题就是:要读取域 n 的数据,就必须首先读取域 $1 \sim n-1$ 域的数据。比如在上面的例子中,要读出第三个子文件的内容,必须首先读到前面两个文件的长度,这样才能知道第三个文件在流中的位置。

现在再回到结构化存储技术问题上。结构化存储内部也是使用流技术,但是它内置了更完善、高效的寻址功能,因此,很好地解决了简单流分域存储的问题。

结构化存储最终生成一个文件。该文件的内部结构和 Windows 的文件系统非常相似。结构化存储文件包含两个概念:存储(Storage)和流(Stream)。存储就相当于文件系统的文件夹,流类似文件夹中的文件;存储可以有子存储。流总是属于某个存储或者子存储。

Windows 提供了一套 API 函数来操纵结构化存储。它们用接口 IStorage 来代表存储,用接口 IStream 来代表流。这些函数和接口被声明在 ActiveX 单元。

API 函数 StgCreateDocfile 用来创建一个结构化存储文件:

```

function StgCreateDocfile(pwcsName: POleStr; grfMode: Longint;
    reserved: Longint; out stgOpen: IStorage): HRESULT; stdcall;

```

其中:

pwcsName: 文件名;
 grfMode: 文件访问方式;
 reserved: 被 Microsoft 保留为以后使用, 应该置为 0;
 stgOpen: 成功创建文件后, 得到它的根存储。

API 函数 StgOpenStorage 用来打开一个存在的结构化存储文件:

```
function StgOpenStorage(pwcsName: POleStr; stgPriority: IStorage;
  grfMode: Longint; snbExclude: TSNB; reserved: Longint;
  out stgOpen: IStorage): HRESULT; stdcall;
```

使用这个函数时, 一般设置 stgPriority=nil、snbExclude=nil、reserved=0。函数执行结果返回根存储 stgOpen。

然后在根存储 stgOpen 中, 可以创建/读取子存储、创建/读取流, 从而完成对一个结构化存储文件的操作。我们已经用简单流分域方法实现了对多个子文件的存取, 现在我们考虑用结构化存储文件 (DocFile) 来实现这个功能。

很简单, 我们用一个存储 (因为只有一个存储, 所以实际上也就是只使用根存储) 和属于这个存储的三个流就可以存储三个子文件了。文件名可以保存在流名中, 文件长度和子文件个数则被存储自动记录了。而一个存储包含的元素 (子存储和流) 可以用一种枚举方法得到。

综上所述, 我们在具体编码之前需要学习几个方法:

(1) 在存储中创建流。使用存储 (IStorage) 的方法:

```
function CreateStream(pwcsName: POleStr; grfMode: Longint;
  reserved1: Longint; reserved2: Longint;
  out stm: IStream): HRESULT; stdcall;
```

其中:

pwcsName: 流名;
 grfMode: 存取方式;
 stm: 返回的流。

(2) 打开一个流。使用存储 (IStorage) 的方法:

```
function OpenStream(pwcsName: POleStr; reserved1: Pointer;
  grfMode: Longint; reserved2: Longint;
  out stm: IStream): HRESULT; stdcall;
```

(3) 存取流 IStream。IStream 提供一些 Write、Read 方法, 但是使用时不是很方便, 所以 Delphi 提供了一个 ToleStream 类来包装 IStream, 方便了流存取。ToleStream 定义在 AxCtrls 单元, 调用以下构造函数就可以使用了, 操作 ToleStream 就是操作 IStream。

```
constructor Create(const Stream: IStream);
```

(4) 枚举一个存储的元素。使用存储 (IStorage) 的方法:

```
function EnumElements(reserved1: Longint; reserved2: Pointer;
  reserved3: Longint; out enm: IEnumStatStg): HRESULT; stdcall;
```

得到一个存储的内部结构接口 IEnumStatStg，然后使用 IEnumStatStg 的方法：

```
function Next(celt: Longint; out elt; pceltFetched: PLongint):
    HRESULT; stdcall;
```

即可实现枚举。其中参数 celt 一般设为 1 (表示一次枚举多少个元素), pceltFetched 一般设为 nil。这个方法返回 TStatStg 类型的枚举结果 elt (elt 中包含元素的名字, 如果元素是流, 那么就是流名, 也就是子文件名)。

好了, 现在可以编码了:

```
uses ActiveX, AxCtrls;
procedure TForm1.FormCreate(Sender: TObject);
begin
    Path := ExtractFilePath(ParamStr(0));
end;

{ 结构化存储三个子文件到一个DocFile }
procedure TForm1.Button3Click(Sender: TObject);
var
    I, Mode: Integer;
    stgRoot: IStorage;
    stmName: String;
    stmData: IStream;
    OleStream: TOleStream;
    LoadStream: TMemoryStream;
begin
    { 访问方式设为: 创建文件、可读可写、拒绝共享 (即独占打开) }
    Mode := STGM_CREATE+STGM_READWRITE+STGM_SHARE_EXCLUSIVE;
    { 创建DocFile: 'lxpbuaa.ss' }
    StgCreateDocfile(
        StringToOleStr(Path + 'lxpbuaa.ss'), Mode, 0, stgRoot);

    { 开始向DocFile 写数据 }
    { LoadStream 用来加载子文件流 }
    LoadStream := TMemoryStream.Create;
    for I := 0 to 2 do
    begin
        case I of
            0: stmName := '乖小孩.gif';
            1: stmName := '浴血长沙 —— 第三次长沙会战守城记.htm';
            2: stmName := '双胞胎.jpg';
        end;
        { 给每个子文件分别创建一个流, 流名即文件名 }
```



```

    stgRoot.CreateStream(StringToOleStr(stmName), Mode, 0, 0, stmData);
    {用包装类TOleStream来操纵IStream}
    OleStream := TOleStream.Create(stmData);
    LoadStream.LoadFromFile(Path + stmName);
    LoadStream.Position := 0;
    {向DocFile的流中写文件内容}
    OleStream.CopyFrom(LoadStream, LoadStream.Size);
    OleStream.Free;
end;
LoadStream.Free;
end;

{从DocFile中分离出原始的三个子文件}
procedure TForm1.Button4Click(Sender: TObject);
var
    Mode: Integer;
    stgRoot: IStorage;
    stmName: PWideChar;
    stmData: IStream;
    OleStream: TOleStream;
    LoadStream: TMemoryStream;
    EnumStatStg: IEnumStatStg;
    StatStg: TStatStg;
begin
    Mode := STGM_READ+STGM_SHARE_EXCLUSIVE;
    {打开一个DocFile}
    StgOpenStorage(StringToOleStr(Path + 'lxpbuaa.ss'),
        nil, Mode, nil, 0, stgRoot);
    LoadStream := TMemoryStream.Create;

    {开始枚举根存储的元素,因为本例的根存储只包含三个流,所以枚举出来的元素就是三个流}
    stgRoot.EnumElements(0, nil, 0, EnumStatStg);
    while EnumStatStg.Next(1, StatStg, nil) = S_OK do
    begin
        {得到元素名,即流名,也即文件名}
        stmName := StatStg.pwcsName;
        {打开枚举到的流}
        stgRoot.OpenStream(stmName, nil, Mode, 0, stmData);
        OleStream := TOleStream.Create(stmData);
        LoadStream.Size := 0;
        LoadStream.CopyFrom(OleStream, OleStream.Size);
    end;
end;

```

```

    { 借助包装类TOleStream和辅助类TMemoryStream, 将流的数据保存到文件 }
    LoadStream.SaveToFile(Path + 'Ss' + stmName);
    OleStream.Free;
  end;
  LoadStream.Free;
end;

```

运行上述程序, 首先按 Button3, 再按 Button4, 我们发现的确生成了结构化文件 lxpbuaa.ss 和三个从 lxpbuaa.ss 分离出的文件: Ss 乖小孩.gif、Ss 双胞胎.jpg、Ss 浴血长沙 ——第三次长沙会战守城记.htm。

9.16 挂钩技术

挂钩 (即大家常常看到的 HOOK) 可以让我们的程序对系统级事件的发生和处理进行控制, 所以作用是十分强大的。如果成功设置了挂钩, 则可以修改甚至屏蔽掉系统事件, 对系统事件进行偷梁换柱, 而接收这些事件的应用程序对此一无所知。因此可以用这种技术来编写一个键盘挂钩, 可以获取任何按键事件, 再加上一些处理代码, 就可以用来获取 QQ 或者信箱密码。

在本节, 我们要讲述如何设置键盘挂钩, 但是不会让它成为一个密码获取器, 大家也不要打密码获取器的主意, 要知道那可是违法的。

API 函数 SetWindowsHookEx 可以用来设置一个挂钩。SetWindowsHook 的功能是一样的, 但是用于 16 位系统, 所以不要再使用它。

SetWindowsHookEx 是这样定义的:

```

function SetWindowsHookEx(idHook: Integer; lpfn: TFNHookProc;
  hmod: HINST; dwThreadId: DWORD): HHOOK; stdcall;

```

各参数含义如下:

idHook: 挂钩类型, 具体取值就不列出了, 大家可以参考 Windows SDK。在本节中, 我们要设置键盘挂钩, 所以取值 WH_KEYBOARD。

lpfn: 挂钩的回调函数。在该函数中可以处理挂钩截获的事件。此函数应该是如下类型:

```

TFNHookProc = function (
  code: Integer; wparam: WPARAM; lparam: LPARAM): LRESULT stdcall;

```

hmod: 回调函数 lpfn 所在 dll 的句柄, 通常用全局变量 HInstance 取得。

dwThreadId: 对那个线程设置挂钩。如果此参数为 0, 则表示将钩住所有线程。

SetWindowsHookEx 返回设置的挂钩句柄。如果该返回值不等于 0, 则表示挂钩设置成功。

对于不同的挂钩类型, 回调函数 TFNHookProc 中的参数和返回值有所不同。如果使用 WH_KEYBOARD 类型, 那么 TFNHookProc 的具体要求如下:

code: 加入小于 0, 表示该事件有多个挂钩, 此时在本挂钩中处理完毕后, 应该调用 API 函数



CallNextHookEx 将事件传递给下一个挂钩继续处理。

wParam：按键的虚拟码。

lParam：按键附加信息，如该键的重复次数、Alt、Ctrl、Shift 键是否按下等。具体可参看 SDK。

下面建立一个实现挂钩的 dll。

在 Delphi 中通过 dll 向导建立一个工程 KeyboardHOOK，然后新建一个单元 HKProc。在 KeyboardHOOK 单元，导出设置挂钩和取消挂钩的两个过程，而这两个过程的实现在 HKProc 中。

KeyboardHOOK 单元：

```
library KeyboardHOOK;

uses
  Windows,
  HKProc in 'HKProc.pas';

{ 输出两个函数 }
exports
  EnableHotKeyHook,      { 供dll加载程序设置挂钩 }
  DisableHotKeyHook;     { 供dll加载程序取消挂钩 }

{$R *.res}

procedure LibraryProc(Reason: Integer);
var
  P: PChar;
begin
  case Reason of
    {dll被加载时，在系统目录创建文件result.txt，我们截获的按键就保存在该文件中，文件变量F定义在HKProc单元}
    DLL_PROCESS_ATTACH:
    begin
      GetMem(P, MAXBYTE);
      GetSystemDirectory(P, MAXBYTE);
      Assign(F, P + '\result.txt');
      Rewrite(F);
      FreeMem(P);
    end;
    {dll卸载时执行HotKeyHookExit过程。此过程在HKProc单元，它取消挂钩并关闭文件F}
    DLL_PROCESS_DETACH: HotKeyHookExit;
  end;
end;
```



```

begin
  { 设置dll的事件处理函数过程为LibraryProc }
  DLLProc := @LibraryProc;
  { 激发dll加载事件, 注意DLL_PROCESS_ATTACH 必须显式激发 }
  LibraryProc(DLL_PROCESS_ATTACH);
end.

```

说明：dll 有 4 个事件，含义如下：

- (1) DLL_PROCESS_ATTACH：dll 在进程中被加载；
- (2) DLL_PROCESS_DETACH：dll 在进程中被卸载；
- (3) DLL_THREAD_ATTACH：dll 在线程中被加载；
- (4) DLL_THREAD_DETACH：dll 在线程中被卸载。

这 4 个事件应该放在 dll 的事件过程中处理。我们再看看 HKProc 单元的代码，此单元的重点是：

- (1) 如何设置键盘挂钩；
- (2) 如何卸载键盘挂钩；
- (3) 在挂钩回调函数中如何处理键盘事件。

```

unit HKProc;

interface

uses
  Windows;

var
  { 定义文件变量 }
  F: File of Char;
  { 设置挂钩函数 }
  function EnableHotKeyHook: BOOL; stdcall;
  { 卸载挂钩函数 }
  function DisableHotKeyHook: BOOL; stdcall;
  { dll 卸载时调用 HotKeyHookExit }
  procedure HotKeyHookExit;

implementation

const
  _KeyPressMask = $80000000 ;

var

```



```

C: Char;
ShiftDown,CapsDown: Boolean;
{ 设置的挂钩句柄}
hNextHookProc: HHook;

{ 挂钩回调函数。在此函数中分离出用户按下的按键，并保存到文件F}
function KeyboardHookHandler(
  iCode: Integer; wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;
begin
  Result := 0;
  { 如果iCode小于0，则表明还有别的挂钩要处理这个消息，所以应该调用下一个挂钩处理该键
    盘事件}
  if iCode < 0 then
  begin
    Result := CallNextHookEx(hNextHookProc,iCode,wParam,lParam);
    Exit;
  end;

  { 以下是对wParam 和lParam参数进行分析，从而得到用户按下的键对应的字符}
  if (lParam and _KeyPressMask) = 0 then { 第32位为0，表示KeyDown状态}
  begin
    ShiftDown := (GetKeyState($10) and _KeyPressMask) = _KeyPressMask;
    CapsDown := (GetKeyState($14) and 1) = 1;

    if wParam < 65 then
    begin
      if ShiftDown then
        C := Chr(wParam-16)
      else
        C := Chr(wParam);
    end else
    begin
      if wParam in [96..105] then
        C := Chr(wParam-48) { 数字键盘}
      else if ShiftDown xor CapsDown then
        C := Chr(wParam)
      else
        C := Chr(wParam+32);
    end;
    { 将键对应的字符写入文件F}
    Seek(F,FileSize(F));
    Write(F,C);
  end;
end;

```



```

    end;
end;

function EnableHotKeyHook: BOOL; stdcall;
begin
    if hNextHookProc = 0 then
    begin
        ReWrite(F);
        { 调用API函数SetWindowsHookEx设置键盘挂钩 }
        hNextHookProc := SetWindowsHookEx(
            WH_KEYBOARD, KeyboardHookHandler, Hinstance, 0);
    end;
    { 是否设置了挂钩 }
    Result := hNextHookProc <> 0;
end;

```

```

function DisableHotKeyHook: BOOL; stdcall;
begin
    if hNextHookProc <> 0 then
    begin
        { 卸载挂钩 }
        UnhookWindowsHookEx(hNextHookProc);
        hNextHookProc := 0;
    end;
    { 是否成功卸载了挂钩 }
    Result := hNextHookProc = 0;
end;

```

```

procedure HotKeyHookExit;
begin
    { 在dll卸载时确保卸载挂钩, 并关闭文件F }
    DisableHotKeyHook;
    Close(F);
end;

end.

```

在 Delphi 的 IDE 中选择菜单 Project | Build KeyboardHOOK, 就编译了 dll 工程并生成了 KeyboardHOOK.dll 链接库。

最后, 新建一个工程 Project1.dpr 来演示如何使用这个 dll。

```
unit Unit1;
```



```
interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  { 定义两个过程类型，对应于挂钩加载和卸载过程 }
  TEnableHotKeyHook = function: BOOL; stdcall;
  TDisableHotKeyHook = function: BOOL; stdcall;

  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

var
  { 定义两个过程变量。过程变量在动态加载dll时被初始化 }
  EnableHotKeyHook: TEnableHotKeyHook;
  DisableHotKeyHook: TDisableHotKeyHook;
  DllHandle: THandle;

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  { Form1 创建时加载挂钩 }
  EnableHotKeyHook;
end;

procedure TForm1.FormDestroy(Sender: TObject);
```

```

begin
  {Form1 销毁时卸载挂钩}
  DisableHotKeyHook;
end;

initialization
  { 动态加载dll }
  { 单元进入时加载dll }
  DllHandle := LoadLibrary('KeyboardHOOK.dll');
  if DllHandle <> 0 then
  begin
    { 初始化过程变量EnableHotKeyHook和DisableHotKeyHook }
    @EnableHotKeyHook := GetProcAddress(DllHandle, 'EnableHotKeyHook');
    @DisableHotKeyHook := GetProcAddress(DllHandle, 'DisableHotKeyHook');
  end;

finalization
  { 单元退出时卸载dll }
  FreeLibrary(DllHandle);

end.

```

注意：对于系统挂钩（即 SetWindowsHookEx 中的 dwThreadId 为 0）必须包含在动态链接库 dll 中，而不能在可执行文件 exe 中完成。因为可执行文件在其他进程中是不可见的。如果是非系统的某特定线程的挂钩（dwThreadId 不为 0），则可以包含在 dll 或者 exe 中。

9.17 TRichEdit 高级开发

TRichEdit 是 Windows 标准控件。我们常常使用它来显示一个文本文件的内容，并可以设置不同段落的字体、颜色、对齐方式等属性，从而将文件内容以非常清晰、鲜明的形式展现给我们。

在本节里，我们用它来显示一个 pas 文件的内容，且显示格式和 Delphi 的 IDE 一样，以展示 TRichEdit 的一些高级用法。

在 TRichEdit 中设置显示格式的原理是：

(1) 首先使用 TRichEdit 的方法：

```

function FindText(const SearchStr: string; StartPos, Length: Integer;
  Options: TSearchTypes): Integer;

```

找到需要特殊显示的字符串的位置。其参数含义如下：

SearchStr：要搜索的字符串；

StartPos：从哪个位置开始搜索；

Length：搜索范围的长度；

Options：搜索选项：

TSearchType = (stWholeWord, stMatchCase);

TSearchTypes = set of TSearchType;

其中，stWholeWord 表示全字匹配，比如在全字匹配条件时，在 'lxpbuaa' 中搜索不到 'lxp'，但是在非全字匹配时，则能搜索到。stMatchCase 表示区分大小写。因为 Delphi 是不区分大小写的，所以在下面的程序中只指定 stWholeWord。

再用属性：**property** SelStart: Integer; 和 **property** SelLength: Integer; 设置选中区域。

(2) 然后用属性：

```
property SelAttributes: TTextAttributes;
```

设置已经选中区域的显示格式。

下面是用 TRichEdit 显示 pas 文件内容的一个例子，运行后显示界面如图 9-4 所示。



图 9-4 pas 文件内容显示程序

从图 9-4 可以看到，编译指令和注释部分用蓝色、斜体显示，字符串常量用蓝色显示，而关键字则用黑体显示。在 TRichEdit 的下方，显示了当前光标所在行和列。

这个程序的代码如下：

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ComCtrls;

type
  TForm1 = class(TForm)
    RichEdit1: TRichEdit;
    Label1: TLabel;
    lbCharPos: TLabel;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure RichEdit1MouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure RichEdit1KeyDown(
      Sender: TObject; var Key: Word; Shift: TShiftState);
  private
    { 关键字列表 }
    KeyWords: TStrings;
    { 将关键字变成黑体 }
    procedure BoldKeyWords(KeyWord: String);
    { 将字符串常量变为蓝色 }
    procedure BlueString;
    { 将编译指令和注释用斜体、蓝色显示 }
    procedure ItalicNoteText;
    { 显示光标所在行号和列号 }
    procedure ShowLineAndCharPos;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

```




```
implementation
```

```
{ $R *.dfm }
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

{ Keywords 中包含了要黑体显示的所有关键字；因为这里只是一个例子，所以还有很多关键字没有列出。实际上，应该用一个外部文件存放所有关键字，程序运行时载入 }

```
Keywords := TStringList.Create;
```

```
with Keywords do
```

```
begin
```

```
  Add('unit');
```

```
  Add('interface');
```

```
  Add('uses');
```

```
  Add('type');
```

```
  Add('function');
```

```
  Add('stdcall');
```

```
  Add('class');
```

```
  Add('procedure');
```

```
  Add('private');
```

```
  Add('public');
```

```
  Add('end');
```

```
  Add('end;');
```

```
  Add('end.');
```

```
  Add('var');
```

```
  Add('implementation');
```

```
  Add('begin');
```

```
  Add('initialization');
```

```
  Add('if');
```

```
  Add('then');
```

```
  Add('finalization');
```

```
end;
```

```
end;
```

```
procedure TForm1.FormDestroy(Sender: TObject);
```

```
begin
```

```
  Keywords.Free;
```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```

    C: Integer;
begin
    { 加载程序所在目录的'Content.pas' 文件内容到RichEdit1中}
    RichEdit1.Lines.LoadFromFile(
        ExtractFilePath(ParamStr(0)) + 'Content.pas');

    { 将关键字黑体显示}
    for C := 0 to KeyWords.Count-1 do
        BoldKeyWords(KeyWords[C]);

    { 将编译指令和注释用斜体、蓝色显示}
    ItalicNoteText;

    { 将字符串常量用蓝色显示}
    BlueString;
end;

{ 将关键字变成黑体}
procedure TForm1.BoldKeyWords(KeyWord: String);
var
    StartPos, FoundAt, SrcLen: Integer;
begin
    StartPos := 0;

    with RichEdit1 do
    repeat
        { 在RichEdit显示的内容中查找关键字}
        SrcLen := Length(Text) - StartPos;
        FoundAt := FindText(KeyWord, StartPos, SrcLen, [stWholeWord]);
        StartPos := FoundAt + Length(KeyWord);

        { 如果找到关键字就变为黑体}
        SelStart := FoundAt;
        SelLength := Length(KeyWord);
        SelAttributes.Style := [fsBold];
    until FoundAt = -1;
end;

{ 将编译指令和注释用斜体、蓝色显示}
procedure TForm1.ItalicNoteText;
var
    StartPos, FoundAt, LastFoundAt, SrcLen: Integer;

```



```

const
  { 编译指令和注释被包含在"{" 中, 对于"{" 注释方式暂没考虑, 不过实现方法是类似的}
  NoteKey1 = '{';
  NoteKey2 = '}' ;
begin
  StartPos := 0;

  with RichEdit1 do
  repeat
    SrcLen := Length(Text) - StartPos;
    LastFoundAt := FindText(NoteKey1, StartPos, SrcLen, []); { 先找 "{" }
    if LastFoundAt <> -1 then
    begin
      StartPos := LastFoundAt + Length(NoteKey1);
      SrcLen := Length(Text) - StartPos;
      FoundAt := FindText(NoteKey2, StartPos, SrcLen, []); { 再找 "}" }

      { 将找到的编译指令和注释内容变为蓝色、斜体 }
      SelStart := LastFoundAt;
      SelLength := FoundAt - LastFoundAt + 1;
      SelAttributes.Style := [fsItalic];
      SelAttributes.Color := clBlue;
      StartPos := FoundAt + Length(NoteKey1)+1;
    end;
  until LastFoundAt = -1;
end;

{ 将字符串常量变为蓝色 }
procedure TForm1.BlueString;
var
  StartPos, FoundAt, LastFoundAt, SrcLen: Integer;
const
  { 字符串常量被包含在两个'"' 中, 这个处理过程和ItalicNoteText 是非常类似的 }
  KeyWord = '""';
begin
  StartPos := 0;

  with RichEdit1 do
  repeat
    SrcLen := Length(Text) - StartPos;
    LastFoundAt := FindText(KeyWord, StartPos, SrcLen, []);
  
```

```

    if LastFoundAt <> -1 then
    begin
        StartPos := LastFoundAt + Length(KeyWord);
        SrcLen := Length(Text) - StartPos;
        FoundAt := FindText(KeyWord, StartPos, SrcLen, []);
        SelStart := LastFoundAt;

        SelLength := FoundAt - LastFoundAt + 1;
        SelAttributes.Color := clBlue;
        StartPos := FoundAt + Length(KeyWord)+1;
    end;
    until LastFoundAt = -1;
end;

procedure TForm1.RichEdit1MouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    { 当通过拖动鼠标实现光标移动时, 显示当前行号和列号 }
    ShowLineAndCharPos;
end;

procedure TForm1.RichEdit1KeyDown(
    Sender: TObject; var Key: Word; Shift: TShiftState);
begin
    { 当通过键盘按键实现光标移动时, 也显示当前行号和列号 }
    ShowLineAndCharPos;
end;

{ 显示当前行号和列号 }
procedure TForm1.ShowLineAndCharPos;
var
    LinePos, CharPos: Integer;
begin
    with RichEdit1 do
    begin
        { 使用消息EM_LINEFROMCHAR取得光标所在行 }
        LinePos := SendMessage(Handle, EM_LINEFROMCHAR, SelStart, 0);
        { 使用消息EM_LINEINDEX取得光标所在列 }
        CharPos := SendMessage(Handle, EM_LINEINDEX, LinePos, 0);
        CharPos := SelStart - CharPos;
    end;
end;

```

```
lbCharPos.Caption := IntToStr(LinePos+1) + ':' + IntToStr(CharPos+1);  
end;  
  
end.
```

因为只是一个例子，所以功能是相当不完整的。在此基础上，本书的所有代码排版工作就是用加强后的“Pas 文件显示器”（它包含剪贴板监视能力，所以只需要在 Word 选中需要排版的代码，然后按 Ctrl+C 键和 Ctrl+V 键就可以完成排版）完成的。

9.18 用 TTreeView 分析数据表的结构

在项目开发中，常常会遇到类似下面的情况：有很多文件资料，这些资料属于不同部门，现在想集中把它们管起来。但是部门分类的层数是不定的，比如有：工程部 | 质量室 | 检查科，后勤部 | 采购室等等。每级部门都有资料要管理。那么你采取什么办法来描述资料的部门归属呢？

你可能想到用多张数据表来描述部门层次，比如表 A 记录一级部门，表 B 记录二级部门……它们逐级构成主从关系。但是我们说了，部门层次可能是不定的，所以无法确定表的数目；而且部门层次越多，表的数目也越多，很显然不好管理。

其实，资料和部门用一张表管理就可以了。在这张表中，专门用一个字段来描述部门。比如该字段的内容可以是：工程部-质量室-检查科。我们用一个分隔符“-”来描述部门层次。在数据浏览和检索时，对该字段使用一定的算法，过滤数据集即可。图 9-5 是一个实现这种功能的程序运行截图。

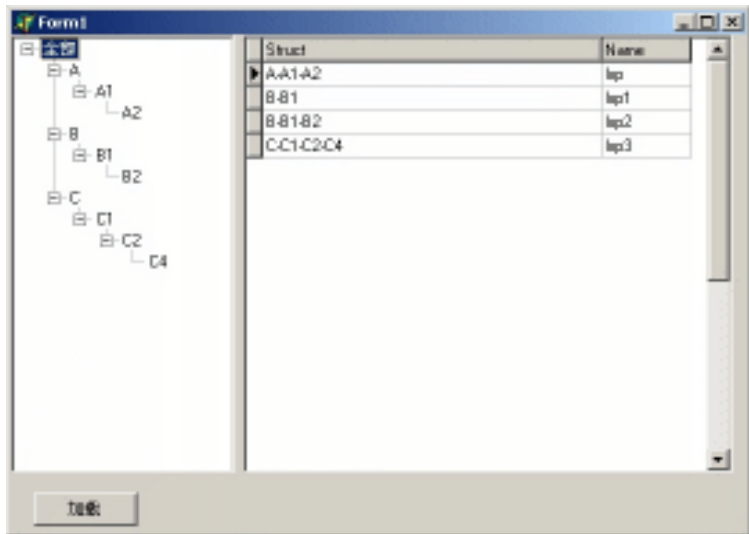


图 9-5 TTreeView 显示记录分类

我们用一个 TTreeView 来显示分类，在 TTreeView 选择不同节点时，TDBGrid 中只显示节点对应的记录。

下面是这个程序的源代码，运行它之前，就首先将光盘对应目录的 TreeViewTable.db 和 TreeViewTable.px 两个文件复制（注意去掉文件的只读属性）到“Common Files\Borland Shared\Data”目录下。

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, DBGrids, DB, DBTables, ComCtrls, StdCtrls;

type
  TForm1 = class(TForm)
    Table1: TTable;
    Table1Struct: TStringField;
    Table1Name: TStringField;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    TreeView1: TTreeView;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    procedure TreeView1Change(Sender: TObject; Node: TTreeNode);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
  procedure LoadNodesFromDataSet(
    TreeView: TTreeView; DataSet: TDataSet; Field: TField);
  procedure FilterDatasetByNode(
    TreeView: TTreeView; DataSet: TDataSet; Field: TField);
  function GetDataSetType(DataSet: TDataSet): Word;
  function GetNodeAllText(TreeView: TTreeView): String;
var
  Form1: TForm1;

implementation
```



```

{ FunAndProc单元定义了一些通用函数和过程，在光盘的lspbuaa.dpk包中}
uses FunAndProc;

const
  { 定义分类字段的分隔符 }
  DeltText = '-';

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  { 从Table1的'Struct' 字段分析出分类并转化为节点显示在TreeView1中 }
  LoadNodesFromDataSet(TreeView1, Table1, Table1.FieldByName('Struct'));
end;

procedure TForm1.TreeView1Change(Sender: TObject; Node: TTreeNode);
begin
  { 用户选中的节点改变时，过滤数据集Table1 }
  FilterDatasetByNode(TreeView1, Table1, Table1.FieldByName('Struct'));
end;

{ 以下是从字段分析节点的算法，写的比较粗糙，部分地方还可以优化 }
procedure LoadNodesFromDataSet(
  TreeView: TTreeView; DataSet: TDataSet; Field: TField);
var
  Strs1, Strs2: TStrings;
  ItemsText, RItemsText: TStringList;
  SavePlace: TBookmark;
  I, C, J: Integer;
  PreNodeEqual: Boolean;
  Stream: TStream;
  procedure AddItems(Strs: TStrings);
  var
    K: Integer;
  begin
    for K := 0 to Strs.Count-1 do
      if Strs[K] <> '' then
        RItemsText.Add(StringOfChar(#9, 1+K) + Strs[K]);
      end;
  end;
begin

```

```
if DataSet = nil then Exit;
ItemsText := TStringList.Create;

with DataSet do
begin
  Open;
  SavePlace := GetBookmark;
  DisableControls;
  ItemsText.BeginUpdate;
  First;
  while not Eof do
  begin
    ItemsText.Add(Field.AsString);
    Next;
  end;
  GotoBookmark(SavePlace);
  FreeBookmark(SavePlace);
  EnableControls;
  ItemsText.EndUpdate;
end;

if ItemsText.Count = 0 then Exit;
RItemsText := TStringList.Create;
RItemsText.Add('全部');
Strs1 := StringToStrings(DeltText, ItemsText[0]);
AddItems(Strs1);
for I := 1 to ItemsText.Count-1 do
begin
  Strs2 := StringToStrings(DeltText, ItemsText[I]);
  C := Strs1.Count;
  if C > Strs2.Count then C := Strs2.Count;
  PreNodeEqual := True;
  for J := 0 to C-1 do
  begin
    PreNodeEqual := PreNodeEqual and (Strs1[J] = Strs2[J]);
    if PreNodeEqual then Strs2[J] := '';
  end;
  AddItems(Strs2);
  FreeAndNil(Strs1);
  FreeAndNil(Strs2);
  Strs1 := StringToStrings(DeltText, ItemsText[I]);
```




```

end;
FreeAndNil(Strsl);
FreeAndNil(ItemsText);

{ 节点信息被保存在RItemsText中, 接下来我们通过一个中间流——Stream加载到
  TreeView中}
Stream := TStringStream.Create(RItemsText.Text);
TreeView.LoadFromStream(Stream);
FreeAndNil(Stream);
FreeAndNil(RItemsText);
{ 全部展开得到的节点}
TreeView.FullExpand;
end;

{ 根据选中的节点过滤数据集}
procedure FilterDatasetByNode(
  TreeView: TTreeView; DataSet: TDataSet; Field: TField);
var
  S: String;
  Node: TTreeNode;
begin
  Node := TreeView.Selected;
  if (Node <> nil) and (DataSet <> nil) and DataSet.Active and
    (Field <> nil) then
  begin
    S := GetNodeAllText(TreeView);
    if S = '' then
    begin
      DataSet.Filtered := False;
      DataSet.Filter := '';
    end else
    begin
      { 不同的数据集类型, 过滤设置Filter要求的格式有所不同}
      case GetDataSetType(DataSet) of
        1,2,5:
          S := ' = ' + QuotedStr(S+'*');
        3,4:
          S := ' Like ' + QuotedStr(S+'%');
      end;
      DataSet.Filter := Field.FieldName + S;
      DataSet.Filtered := True;
    end;
  end;
end;

```



```

    end;
  end;
end;

function GetNodeAllText(TreeView: TTreeView): String;
var
  CurrNode: TTreeNode;
begin
  CurrNode := TreeView.Selected;
  while (CurrNode <> nil) and (CurrNode.Level > 0) do
  begin
    Result := CurrNode.Text + DeltText + Result;
    CurrNode := CurrNode.Parent;
  end;
  Result := Copy(Result, 1, Length(Result)-Length(DeltText));
end;

function GetDataSetType(DataSet: TDataSet): Word;
var
  Name: String;
begin
  Name := DataSet.ClassName;
  if Name = 'TTable' then Result := 1
  else if Name = 'TQuery' then Result := 2
  else if Name = 'TADOTable' then Result := 3
  else if Name = 'TADOQuery' then Result := 4
  else if Name = 'TClientDataSet' then Result := 5
  else Result := 0;
end;

end.

```

9.19 SQL 语句分析器

在本节中，我们来做一个简单的 SQL 语句分析器，其原理就是在 SQL 语句中查找 SQL 语法定义的关键字，从而将整个的 SQL 语句分为不同段落，如 SELECT、FROM、WHERE 等。这样可以方便对 SQL 语句进行变化，这在数据查询、排序等时候是非常方便的。

因为这个分析器比较简单，所以只能包含几个常用的关键字：SELECT、FROM、WHERE、GROUP BY 和 ORDER BY。

含有其他关键字的 SQL 语句请不要使用这个分析器分析，否则可能发生错误。这个分析器的代码



如下：

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  { TArrayResult和TSQL两个类型由分析器使用,TSQL用来保存分析结果}
  TArrayResult = Array[0..1] of Integer;
  TSQL = record
    SQL,           { 整个语句}
    Fields,        { 字段}
    TableName,     { 表名}
    Condition,     { 查询条件}
    OrderBy,       { 排序字段}
    GroupBy: String; { 分组字段}
  end;

  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

uses FunAndProc;

{$R *.dfm}

{ 分析器主体。它要求的参数是TQuery.SQL或者TADOQuery.SQL等}
function GetSubSQL(SQL: TStrings): TSQL;

```



```

var
  I,J: Integer;
  KeyList: TStrings;
  S,S1: String;
  R: TSQL;
function GetByPos(Text: String; Mode: Word): TArrayResult;
var
  I,J,K: Integer;
  Key1,Key2,S: String;
begin
  case Mode of
    1: Key1 := ' GROUP ';
    2: Key1 := ' ORDER ';
  else Exit;
  end;
  Key2 := ' BY ';

  I := Pos(Key1,Uppercase(Text));
  K := 0;
  if I > 0 then
  begin
    J := I + Length(Key1)-1;
    S := Copy(Text,J,Length(Text)-J+1);
    Key2 := ' BY ';
    K := Pos(Key2,Uppercase(S));
    if K = 0 then I := 0
    else Inc(K,Length(Text)-Length(S)+Length(Key2));
  end;
  Result[0] := I;
  Result[1] := K;
end;
begin
  { 首先去掉SQL中的控制符，并转化为一个字符串}
  S := Trim(StringsToString(Chr($20),SQL));

  KeyList := TStringList.Create;
  KeyList.Add(' SELECT ');
  KeyList.Add(' FROM ');
  KeyList.Add(' WHERE ');

  { 字段}

```

```

I := Length(KeyList[0]) + 1;
J := Pos(KeyList[1],UpperCase(S));
R.Fields := Trim(Copy(S,I,J-I));
{表名}
I := Pos(KeyList[1],UpperCase(S)) + Length(KeyList[1]);
S1 := TrimLeft(Copy(S,I,Length(S)));
J := Pos(' ',S1);
if J > 0 then S1 := Trim(Copy(S1,1,J-1))
else S1 := Trim(S1);
R.TableName := S1;
{条件}
I := Pos(KeyList[2],UpperCase(S));
if I > 0 then
begin
  Inc(I,Length(KeyList[2]));
  J := GetByPos(S,1)[0];
  if J = 0 then J := GetByPos(S,2)[0];
  if J > 0 then S1 := Trim(Copy(S,I,J-I))
  else S1 := Trim(Copy(S,I,Length(S)));
end else S1 := '';
R.Condition := S1;
{分组字段}
I := GetByPos(S,1)[1];
if I > 0 then
begin
  J := GetByPos(S,2)[0];
  if J > 0 then S1 := Trim(Copy(S,I,J-I))
  else S1 := Trim(Copy(S,I,Length(S)));
end else S1 := '';
R.GroupBy := S1;
{排序字段}
I := GetByPos(S,2)[1];
if I > 0 then S1 := Trim(Copy(S,I,Length(S)))
else S1 := '';
R.OrderBy := S1;
{全部}
R.SQL := 'SELECT ' + R.Fields + ' FROM ' + R.TableName;
if R.Condition <> '' then R.SQL := R.SQL + ' WHERE ' + R.Condition;
if R.GroupBy <> '' then R.SQL := R.SQL + ' GROUP BY ' + R.GroupBy;
if R.OrderBy <> '' then R.SQL := R.SQL + ' ORDER BY ' + R.OrderBy;
FreeAndNil(KeyList);

```

```
Result := R;
end;

{ 显示分析器的分析结果 }
procedure TForm1.Button1Click(Sender: TObject);
var
    SQL: TStrings;
    R: TSQL;
    S: String;
begin
    SQL := TStringList.Create;
    SQL.Add(
        'SELECT name,sex FROM BlueSky WHERE age>100 GROUP BY sex ORDER BY age' );
    R := GetSubSQL(SQL);
    with R do
        S := 'ALL:' + R.SQL + #13 +
            'SELECT:' + Fields + #13 +
            'FROM:' + TableName + #13 +
            'WHERE:' + Condition + #13 +
            'GROUP BY:' + GroupBy + #13 +
            'ORDER BY:' + OrderBy;
    ShowMessage(S);
    FreeAndNil(SQL);
end;

end.
```

分析结果如图 9-6。

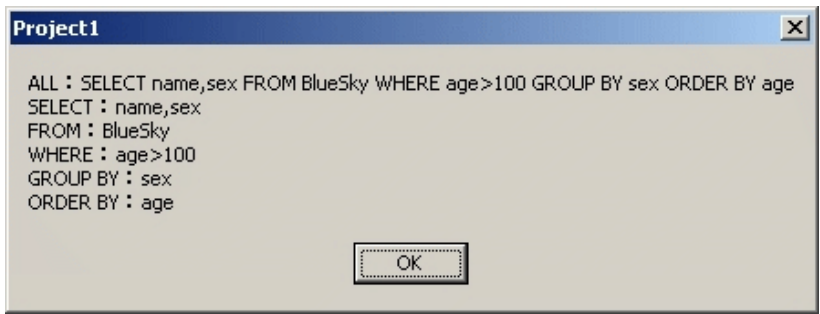


图 9-6 SQL 分析器使用示例

在分析器 GetSubSQL 的基础上，可以再定义一个函数，用来合成新的 SQL 语句。比如：

```

function SetSQL(SQL: TSQL): String;
begin
  with SQL do
  begin
    Result := 'SELECT ' + Fields + ' FROM ' + TableName;
    if Trim(Condition) <> '' then Result :=
      Result + ' WHERE ' + Trim(Condition);
    if Trim(GroupBy) <> '' then Result :=
      Result + ' GROUP BY ' + Trim(GroupBy);
    if Trim(OrderBy) <> '' then Result :=
      Result + ' ORDER BY ' + Trim(OrderBy);
  end;
end;

```

我们可以先使用 GetSubSQL 分析原来的 SQL 语句，得到一个 TSQL 类型的分析结果，然后改变 TSQL 的一些字段，最后调用函数 SetSQL 合成。

9.20 剪贴板高级编程

剪贴板是 Windows 的一块内存区，使用它可以在不同界面、不同应用程序间实现信息共享。Windows 提供了一套 API 函数来操纵剪贴板。

VCL 的一些类有各自的剪贴板操纵方法，如：

(1) 将数据剪贴到剪贴板：CutToClipboard。

TCustomEdit、TCustomMaskEdit、TCustomMemo、TDBImage 等类有此方法。

(2) 将数据复制到剪贴板：CopyToClipboard。

TCustomEdit、TCustomMemo、TCustomTextViewer、TDBImage、TDdeServerItem 等类有此方法。

(3) 从剪贴板复制数据：PasteFromClipboard。

TCustomEdit、TCustomMaskEdit、TCustomMemo、TDBImage 等类有此方法。

更通用的是，Delphi 在 Clipbrd 单元定义了一个类 TClipboard，它对剪贴板 API 函数进行了封装。并且定义了一个全局变量 Clipboard 来代表剪贴板。因此，如果一个应用程序引用了 Clipbrd，那么就可以直接使用 Clipboard 来操纵剪贴板。下面我们来详细学习 TClipboard 类。

TClipboard 的属性：

```
property AsText: string;
```

将剪贴板的数据转化为一个字符串。

这个属性可以用来从剪贴板获得数据，也可以将数据复制到剪贴板。比如：

```

uses Clipbrd;
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);
begin
    Clipboard.AsText := 'lxpbuaa';
    ShowMessage(Clipboard.AsText);
end;
property Formats[Index: Integer]: Word;

```

用来获取剪贴板上的数据格式。剪贴板上可以同时放置不同格式的数据，例如，我们在一个 Word 文档中写了一些文字并插入了图片，然后选中所有文字和图片，最后按 Ctrl+C 键将这些数据复制到剪贴板上。此时，剪贴板包含两种格式的数据：文本和图片。

常用的数据格式有以下一些：

CF_TEXT	文本
CF_BITMAP	位图
CF_DIB	带调色板的位图
CF_METAFILEPICT	媒体文件
CF_PALETTE	调色板
CF_PICTURE	TPicture 类型对象
CF_COMPONENT	TPersistent 类型对象

我们在 Windows 单元搜索“CF_”可以看到所有预定义的剪贴板数据格式。我们可以用 API 函数 RegisterClipboardFormat 来注册新的数据类型。

```
property FormatCount: Integer;
```

剪贴板上当前数据格式数。

TClipboard 的方法：

```
procedure Assign(Source: TPersistent);
```

将一个 TPersistent 类型的对象复制到剪贴板，很明显，这时候的数据格式是 CF_COMPONENT。比如可以用下面两行语句将 Image1 的图形复制到 Image2：

```

begin
    Clipboard.Assign(Image1.Picture);
    Image2.Picture.Assign(Clipboard);
end;

procedure Clear;

```

清除剪贴板上的所有数据。



```
procedure Open;
```

打开剪贴板，此时为独占方式，即在调用 Close 方法前，别的程序无法改变剪贴板上的数据。

Open 和 Close 方法一般用在需要向剪贴板添加多项数据的时候。如果只添加一项数据，则没有必要使用 Open 和 Close。

```
procedure SetTextBuf(Buffer: PChar);
```

复制文本数据到剪贴板。

```
function GetTextBuf(Buffer: PChar; BufSize: Integer): Integer;
```

从剪贴板取得文本数据。BufSize 指定需要取得多少长度的数据。

```
procedure SetComponent(Component: TComponent);
```

复制 TComponent 对象到剪贴板。

```
function GetComponent(Owner, Parent: TComponent): TComponent;
```

从剪贴板取得 TComponent 对象，并指定该对象的拥有者和父控件。

但是要注意的是，调用 GetComponent 前首先要用 RegisterClasses 过程注册该类。如：

```
begin
```

```
  { 将Memo1复制到剪贴板 }
```

```
  Clipboard.SetComponent(Memo1);
```

```
  { 销毁Memo1，此时窗体上没有了Memo1了 }
```

```
  Memo1.Free;
```

```
  { 注册类TMemo }
```

```
  RegisterClasses([TMemo]);
```

```
  { 从剪贴板取得Memo1，并知道它的拥有者和父控件都是Form1，这样Memo1又在Form1上出现了 }
```

```
  Memo1 := Clipboard.GetComponent(Self, Self) as TMemo;
```

```
  Memo1.Left := 0;
```

```
  { 注销类TMemo }
```

```
  UnRegisterClasses([TMemo]);
```

```
end;
```

```
procedure HasFormat(Format: Word): Boolean;
```

判断剪贴板当前是否包含 Format 格式的数据。例如：

```
begin
```

```
  Clipboard.SetComponent(Memo1);
```

```
  if Clipboard.HasFormat(CF_COMPONENT) then
```

```
    ShowMessage('有CF_COMPONENT格式的数据');
```

```
end;
```

```
procedure SetAsHandle (Format: Word; Value: THandle);
```

将句柄 Value 对应的数据按格式 Format 保存到剪贴板上。

注意使用 `SetAsHandle` 后，不要用代码销毁 `Value`，因为此时剪贴板已经接管了它。

```
function GetAsHandle (Format: Word): THandle;
```

取得剪贴板上 `Format` 格式数据的句柄。

这两个方法通常用来在剪贴板上存取自定义格式的数据。

在本节的最后部分，我们用一段代码演示如何自定义剪贴板数据格式。

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls;

type
  { 定义记录类型TDataRecord来管理自定义数据 }
  TDataRecord = packed record
    Name,
    Sex,
    Hobby: String[10];
  end;

  TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    Edit1: TEdit;
    Label2: TLabel;
    Edit2: TEdit;
    Label3: TLabel;
    Edit3: TEdit;
    Memo1: TMemo;
    Button2: TButton;
    Label4: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```



```
var
    Form1: TForm1;
    CF_CUSTOMFORMAT: Word;

const
    CF_CUSTOMFORMATNAME = 'CF_CUSTOMFORMAT';

implementation

uses Clipbrd;

{$R *.dfm}

{ 将自定义数据复制到剪贴板 }
procedure TForm1.Button1Click(Sender: TObject);
var
    DataRec: TDataRecord;
    Data: THandle;
    PData: Pointer;
begin
    with DataRec do
    begin
        Name := Edit1.Text;
        Sex := Edit2.Text;
        Hobby := Edit3.Text;
    end;

    { 在全局堆 (global heap) 上分配一块内存区, 返回值Data为该内存区的句柄 }
    Data := GlobalAlloc(GMEM_MOVEABLE, SizeOf(TDataRecord));
    { 获得该内存区的指针PData }
    PData := GlobalLock(Data);
    { 将记录DataRec的内容复制到指针PData指向的内存区Data }
    Move(DataRec, PData^, SizeOf(TDataRecord));
    { 接下来, 将内存区Data的数据以CF_CUSTOMFORMAT格式存放到剪贴板上 }
    Clipboard.SetAsHandle(CF_CUSTOMFORMAT, Data);
    GlobalUnlock(Data);
    { 注意千万不要使用下一句, 因为此时Data已经交由剪贴板管理 }
    { GlobalFree(Data); }
end;

{ 从剪贴板取出CF_CUSTOMFORMAT格式的数据 }
procedure TForm1.Button2Click(Sender: TObject);
```

```
var
    DataRec: TDataRecord;
    Data: THandle;
    PData: Pointer;
begin
    { 取得CF_CUSTOMFORMAT 格式数据的句柄 }
    Data := Clipboard.GetAsHandle(CF_CUSTOMFORMAT);
    if Data = 0 then Exit;
    { 取得Data对应的指针 }
    PData := GlobalLock(Data);
    { 将指针PData对应的数据复制到记录DataRec }
    Move(PData^, DataRec, GlobalSize(Data));
    GlobalUnlock(Data);
    { 显示取得的数据 }
    with DataRec do
        ShowMessage(Name + #13#10 + Sex + #13#10 + Hobby);
end;

initialization
    { 单元初始化时,调用API函数RegisterClipboardFormat,注册名为CF_CUSTOMFORMATNAME
      的剪贴板格式,格式常数返回到CF_CUSTOMFORMAT }
    CF_CUSTOMFORMAT := RegisterClipboardFormat(CF_CUSTOMFORMATNAME);

end.
```

上述程序运行界面如图 9-7。

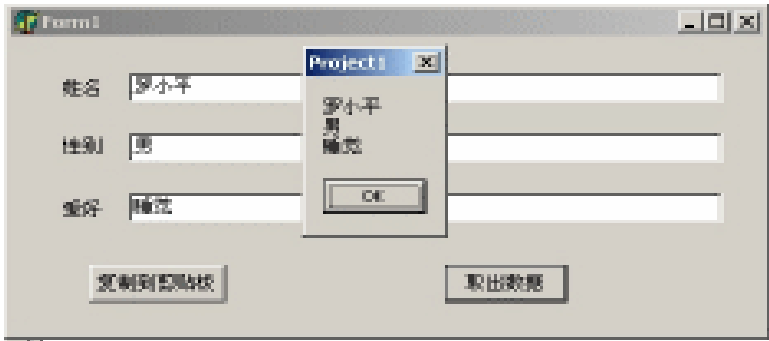


图 9-7 自定义剪贴板数据格式

9.20
剪贴板高级编程

第 10 章 综合例子

——使用 Socket 传输多个文件

本章讲述了 Delphi 在开发网络程序方面的一些应用。

TServerSocket 和 TClientSocket 是 Delphi 开发网络程序的两个基础性组件。使用这两个组件，可以开发聊天程序、下载程序等各种网络应用程序。在本书的最后一章，我们采用这两个组件来开发文件下载的服务端和客户端程序，并在此基础上封装了 TlxpServerSocketForFiles 和 TlxpClientSocketForFiles 两个组件，这两个组件可以实现文件自动下载。

本章的内容是一个综合实例，较全面地运用了本书前面所讲的知识，算是奉献给读者朋友的一道大餐，以感谢朋友们购买和阅读本书。

10.1 Socket 简介

Internet 上有大量的服务器提供众多的服务，如：HTTP、FTP、SMTP、Telnet、Time 和 IPX 等。每种服务都是在一个专用的公开端口上实现的，因此，别的计算机可以使用特定的协议连接服务提供者的公开端口，从而获取特定服务。它和日常生活中的服务概念是相似的，如银行提供存贷款服务、公交公司提供交通运输服务。

因为有一些服务种类已经存在很长时间，并且为网络世界的信息交互作出了巨大贡献，为了让更多的人知道并享受这些服务，它们所使用的端口就逐渐被固定下来并广为人知，如：

HTTP	80
FTP	21
SMTP	25
Telnet	23
Time	37
IPX	213

在 Windows 系统中，266 以下的端口值被保留使用，1024 以下的很多端口也已经被占用。因此，在开发我们自己的网络服务程序时，一般应该使用大于 1024 的端口。

在命令行键入：netstat -a，可以看到本机的端口占用情况列表。

网络上两个程序为了相互通讯运行，构成服务端/客户端结构，连接的每一端可称为一个 Socket（或者套接字）。客户程序可以向服务端 Socket 发送请求，服务端收到后处理此请求，然后将处理结果发送给客户端 Socket，从而形成一次应答。如此重复必要次数，就完成了一次通讯。

Socket 是一种底层连接。客户机和服务器通过写入和读取 Socket 的字节流进行通信。双方的写和读必须遵守共同的协议，也就是说，通过 Socket 相互传送信息时所用的语言必须是事先协定好的。

目前的各种常用通讯协议，如 HTTP、FTP、Finger、Time 等，是实现在 Socket 的底层连接之上的。

可以讲 Socket 是一系列为了实现底层连接而定义的规则。在 Windows 操作系统中，这一系列规则实现出来后就是一系列 dll，或者说 API 函数库。

我们可以直接调用这些 dll 中的函数，如：socket、listen、connect 等来构建通讯程序。但是大家知道，直接使用 API 函数来构筑应用程序是非常麻烦的。于是出现了大量组件，它们封装了这些底层函数，大大提高了程序开发效率，如 Delphi 的 TServerSocket、TClientSocket、TUdpSocket 以及 TIdTCPServer 和 TIdTCPClient 等。

TServerSocket、TClientSocket 是两个功能强大的组件。在本章，我们采用这两个组件来完成多文件传输功能。

小结

Socket 是网络传输技术中一个非常重要的概念，本节简要介绍了 Socket 的概念和用途，希望能对读者朋友理解 Socket 有所帮助。

10.2 TServerSocket 和 TClientSocket

TServerSocket 和 TClientSocket 两个组件在 Delphi 的 Internet 组件页。TServerSocket 用来管理 TCP/IP 服务端的套接字连接，相应地，TClientSocket 管理客户端套接字连接。

TServerSocket 比较重要的属性有如下一些：

property Active: Boolean;	TServerSocket 是否已经启动，启动后自动在指定端口侦听，等待别的程序连接它。
property Port: Integer;	在哪个端口侦听。
property Service: string;	服务的描述。一般情况下可以设为空；如果是“FTP”、“HTTP”、“Finger”、“Time”等公开的协议名，实际侦听端口会被自动指定为这些公开协议默认的端口。
property ServerType: TServerType;	

其中：TServerType = (stNonBlocking, stThreadBlocking); 用于指定线程模式。

TServerSocket 和客户端建立连接后，会自动产生线程实现读写。

stNonBlocking 表示单线程执行。即所有连接都是在一个线程中完成读写；单线程模式也称为同步模式、非阻塞模式。

使用同步模式时，应该在事件 OnClientRead 中完成数据写入（即向客户端发送数据），在 OnClientWrite 中完成数据读出（即读取客户端发来的数据）。

stThreadBlocking 表示多线程执行。即每个连接都有自己单独的线程来完成读写，也称为异步模式、阻塞模式。当一个客户端请求连接时，会自动触发事件 OnGetThread，我们应该在 OnGetThread 中为该客户端的连接建立单独的线程，并将创建的线程赋给参数 SocketThread。

使用异步模式时，数据读写应该在线程中建立 TWinSocketStream 的实例来完成，而不是在事件 OnClientRead 和 OnClientWrite 中。

TClientSocket 比较重要的属性有如下一些：

```
property Active: Boolean;
property Port: Integer;
property Service: string;
```

以上三个属性含义同 TServerSocket。

```
property Address: string;
property Host: string;
```

以上两个属性用于指定服务端所在的计算机。Address 用 IP 地址表示，Host 用计算机名表示。如果 Address 和 Host 都设置了值，则只有 Host 有效。

```
property ClientType: TClientType;
```

其中：TClientType = (ctNonBlocking, ctBlocking);

ClientType 仍然是用来指定线程模式。

ctNonBlocking：非阻塞模式或异步模式。此时在事件 OnRead 和 OnWrite 进行读写过程中，主应用程序的执行被阻塞，比如此时鼠标、键盘事件不会得到响应。

ctBlocking：阻塞模式或同步模式。主应用程序可以得到同步执行。此时应该用类 TWinSocketStream 创建实例来完成读写。

使用 ctNonBlocking 方式有一个缺陷，那就是：在 OnRead 事件中并不总能读到从服务端送来的最后 1Bit 数据，换句话说，服务端发送的最后 1Bit 数据并不总能激发一个 OnRead 事件。所以使用这种模式时，一般都要在 OnDisconnect 中试着读一次数据，以确保最后 1Bit 数据不被遗漏。

最后我们举个简单的例子：

```
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
    Forms, Dialogs, ScktComp;

type
    TForm1 = class(TForm)
        ClientSocket1: TClientSocket;
        ServerSocket1: TServerSocket;
    procedure FormCreate(Sender: TObject);
    procedure ServerSocket1ClientConnect(Sender: TObject;
        Socket: TCustomWinSocket);
    procedure ClientSocket1Read(
```

```

    Sender: TObject; Socket: TCustomWinSocket);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

    {$R *.dfm}

    { 局部函数Get_ComputerName用来取得本地计算机 }
    function Get_ComputerName: String;
    var
        iSize: LongWord;
        ComputerName: PChar;
    begin
        iSize := MAX_COMPUTERNAME_LENGTH + 1;
        GetMem(ComputerName, iSize);
        { GetComputerName 是一个API函数, 用于取得计算机名字 }
        GetComputerName(ComputerName, iSize);
        Result := ComputerName;
        FreeMem(ComputerName);
    end;

    procedure TForm1.FormCreate(Sender: TObject);
    begin
        with ServerSocket1 do
            begin
                Port := 5643;
                Open;    { 或者Active := True; }
            end;
        with ClientSocket1 do
            begin
                Port := 5643;
                Host := Get_ComputerName;
                Open;    { 或者Active := True; }
            end;
    end;

```




```

end;

procedure TForm1.ServerSocket1ClientConnect(
  Sender: TObject; Socket: TCustomWinSocket);
begin
  { 客户端连接时向它问好 }
  Socket.SendText(' ClientSocket1你好! ');
end;

procedure TForm1.ClientSocket1Read(
  Sender: TObject; Socket: TCustomWinSocket);
begin
  { 读取从服务端发来的问候 }
  ShowMessage(Socket.ReceiveText);
end;

```

小结

本节详细介绍了 TServerSocket 和 TClientSocket 组件的用途和重要属性。

本节的重点是理解 TServerSocket 和 TClientSocket 的线程模式及其使用方法。

10.3 设计通讯协议

本章需要实现的功能是：建立一个文件下载服务端和文件下载客户端；服务端可以同时应答多个客户端并向客户端提供下载多个文件服务。

因此在服务端，TServerSocket 应该采用 stThreadBlocking 模式，利用 TWinSocketStream 实现数据读写；客户端可以采用 ctNonBlocking 和 ctBlocking 中的任何一种，但是我们讲了 ctNonBlocking 模式的一个缺陷，如果采用 ctNonBlocking 模式，则必须在 OnDisconnect 事件中完成最后字节数据的读取，否则采用 ctBlocking 模式利用 TWinSocketStream 来进行读写。

为了统一和简便，在服务端采用 stThreadBlocking 模式；在客户端采用 ctBlocking 模式并使用 TWinSocketStream 进行读写；并且服务端和客户端的 TWinSocketStream 读写都放在线程中实现。

首先来看看 TWinSocketStream 的几个重要方法：

```
constructor Create(ASocket: TCustomWinSocket; Timeout: Longint);
```

构造函数。服务端和客户端建立连接后，每端都获得一个 Socket 对象，参数 ASocket 就是这个 Socket。此构造函数的意思是在 Socket 基础上建立一个连接流对象来完成数据传输。Timeout 为超时值，如果在 Timeout 时间内，一个读或写操作没有完成，则自动放弃。

```
function WaitForData(Timeout: Longint): Boolean;
```

在 Timeout 时间内等待连接准备到位。所有读写操作应该在此函数返回 True 时进行。

```
function Read(var Buffer; Count: Longint): Longint; override;
```

覆盖了 TStream 抽象类的 Read 方法, 请求从连接流中读取 Count 字节的数据到缓冲区 Buffer (因此, 缓冲区的大小应该不小于 Count 字节) 中, 返回值是实际读取的字节数。返回值总是不大于 Count。

```
function Write(const Buffer; Count: Longint): Longint; override;
```

覆盖了 TStream 抽象类的 Write, 请求将缓冲区的 Count 字节数据吸入连接流, 返回值是实际写入的字节数。返回值总是不大于 Count。

TStream 类还提供了 ReadBuffer 和 WriteBuffer 方法, 功能与 Read、Write 是相同的。不过不同的是, 如果返回值不等于 Count, 则 Read 和 Write 会触发异常, 而 ReadBuffer 和 WriteBuffer 不会。

大家知道, Windows 默认缓冲区的大小是 4K, 所以 Count 一般不应该大于 4K, 否则一次调用 Read 和 Write 实际上要多次才能完成。

好, 了解了 TWinSocketStream 读写数据的基本方法, 就可以来设计服务端和客户端通讯的协议了。

我们是用 Socket 连接来传输文件, 大家知道, 一个文件的大小往往都是大于 Windows 默认缓冲区 4K 的。因此, 一个文件流的传输要分很多次 (即至少需要文件/4KB 次) 才能完成。服务端向客户端发送第 n 条数据后, 也不应该马上发送第 $n+1$ 条, 因为此时客户端还不一定已完成第 n 条数据的读取, 否则可能造成数据拥塞; 同样, 客户端在读取了第 n 条数据后, 也不应该马上读取第 $n+1$ 条, 因为此时服务端还不一定已完成第 $n+1$ 条数据。所以, 我们在服务端和客户端之间建立一个“抛球”规则, “球”相当于一个令牌, 任何一端在没有获得这个令牌时, 不应该进行下一次的数据操作。

简单一点地说, 可以划分为以下步骤:

- (1) 服务端首先在提供计算机的某个端口启动, 开始侦听任何客户端的连接请求。
- (2) 客户端向服务端请求连接。
- (3) 服务端响应连接, 并建立一个连接线程。并告知客户端“连接成功”(“连接成功”就相当于一个令牌)。
- (4) 客户端获得“连接成功”令牌, 向服务端“请求数据传输”(这也可以看做是一个令牌)。
- (5) 服务端获得“请求数据传输”令牌, 开始从文件流中提取第 1 条数据并发出。
- (6) 客户端接收第 1 条数据, 然后再次“请求数据传输”。
- (7) 服务端获得“请求数据传输”令牌, 开始发送第 2 条数据。

重复 (5)(6) 步骤……最终可以完成一个完整的数据传输过程。

在这里, 我们所说的“通讯协议”, 就是指以上步骤和令牌, 而不是说 TCP、UDP 等协议概念。

在上面的步骤中, 还存在一个悬而未决的问题, 那就是客户端怎么知道在接收了多少条数据后才知文件的所有数据已经接收完毕, 否则它将一直“WaitForData”, 永远不能完成一次文件传输。

一般有两种行之有效的方法:

- (1) 在服务端开始传输文件之前, 首先告知客户端文件的大小。这样, 客户端在每次接收一条数据后进行数据大小累加, 当累加结果等于事先知道的文件大小时, 该文件就传输完了。
- (2) 服务端传输最后一条数据后, 例行公式的等待客户端获取下一条数据的请求, 收到请求后,

发送一段约定的数据（通常是一段文本），如“`AtEndOfFile`”。客户端收到约定文本后，也能知道文件传输结束，然后断开连接。

第二种方法存在一个弱点，那就是：某一段文件数据可能就是这样的约定文本，或者二者的二进制编码相同，这样一来，客户端在接收到这样的文件数据段后，会误认为文件传输已经结束，从而中断连接，但它认为的“约定文本”实际乃是文件的内容而非真的“约定文本”，这就违背了服务端的本意，最终造成文件传输失败。这种情况发生的可能性是很小的，其发生条件是：

（1）文件数据中存在这样的数据段。

（2）服务端正好能将“约定数据”作为一段完整数据发出。因此，“约定数据”在文件流中的开始位置正好是缓冲区大小的整数倍。

实际开发中，缓冲区的大小一般都是在 1K、2K、3K、4K 几个值中选择，太小了没有必要，一般也不应该大于 Windows 默认缓冲区大小 4K。而真正的约定文本也不可能长达 K 级，因此，进一步将条件（2）逼到“‘约定数据’正好是文件流中的最后一笔数据，且开始位置正好是缓冲区大小的整数倍”的地步。这样一来，同时满足两个条件的情况几乎没有可能发生。

尽管这种情况发生的可能性是极小的，但是一旦发生后就是致命的。所以在开发正式程序时，我们一定要采用第一种方法。

通过一系列分析，可以开始设计详细的通讯协议了。大家可以自己规划一下，我最终的设计结果如下：

```
const
  DefaultPort = 5643;           { 服务器缺省侦听端口 }
  KEY_Clt: Array[1..4] of String = { 从客户端发出以下令牌 }
    ('AskForFileName',         { 请求文件名 }
     'AskForFilesLength',      { 请求文件长度 }
     'AskForFilesData',        { 请求发送文件 }
     'WanttoDisconnect');      { 文件发送完成，告知服务端连接可以关闭了 }
  KEY_Srv: Array[1..2] of String = { 从服务端发出以下令牌： }
    ('Return1', { 后面跟的是所有文件名，文件名之间用FileNameSepStr分隔 }
     'Return2'); { 后面跟的是所有文件长度，文件长度之间用FilesLengthSepStr分隔 }
  FileNameSepStr = '|';
  FilesLengthSepStr = ',';
```

使用上述常数，可以定义服务端和客户端的交互过程如下：

（1）Server 在 DefaultPort 开始侦听。

（2）Client 向 Server 请求连接，并建立客户端线程：ClientThread。

ClientThread 发送“AskForFileName”请求。

Server 收到连接请求，建立服务端线程：ServerThread。

ServerThread 收到请求，发送 “Return1” + 文件名 1 + “|” + ... + 文件名 n。

(3) ClientThread 收到回应，发送 “AskForFilesLength” 请求。

ServerThread 收到请求，发送 “Return2” + 文件长度 1 + “,” + ... + 文件长度 n。

(4) ClientThread 收到回应，发送 “AskForFilesData”。

ServerThread 收到请求，发送第一条数据。

(5) ClientThread 收到回应，第二次发送 “AskForFilesData”。

ServerThread 收到请求，发送第二条数据。

.....

(n) ClientThread 发现全部文件接收完毕，发送 “WanttoDisConnect” 并结束线程。

ServerThread 收到请求，关闭该连接，结束线程。

这样，一次发送多文件的过程就完成了。

小结

本节的重点：

(1) 辅助类 TWinSocketStream 的几个重要方法。

(2) 如何设计稳定可靠的文件传输通讯协议。

10.4 实现服务端

根据上一节的协议设计，我们开发出服务端，其代码如下：

公用库文件（定义了服务端和客户端使用的令牌，客户端也要使用此文件）：

```
unit FunAndProc;

interface

uses Windows, Classes, SysUtils;

const
  DefaultPort = 5643;           { 服务器缺省侦听端口 }
  KEY_Clt: Array[1..4] of String = { 从客户端发出以下令牌 }
    ( 'AskForFileName',         { 请求文件名 }
      'AskForFilesLength',      { 请求文件长度 }
      'AskForFilesData',       { 请求发送文件 }
      'WanttoDisConnect' );    { 文件发送完成，告知服务端连接可以关闭了 }
  KEY_Srv: Array[1..2] of String = { 从服务端发出以下令牌： }
    ( 'Return1',               { 后面跟的是所有文件名，文件名之间用FileNameSepStr分隔 }
```



```

    'Return2');      {后面跟的是所有文件长度，文件长度之间用FilesLengthSepStr
                      分隔}
    FileNameSepStr = '|';
    FilesLengthSepStr = ',';

    {StringToStrings将一个字符串转化为字符串列表，转化方法由字符串中的分隔符SepStr决定}
    function StringToStrings(SepStr: String; S: String): TStrings;

    {将字符串列表转化为字符串，由SepStr分隔}
    function StringsToString(SepStr: String; Strs: TStrings;
        GetFileName: Boolean = False): String;

    {返回本机的名字}
    function Get_ComputerName: String;

implementation

function StringToStrings(SepStr: String; S: String): TStrings;
var
    P: Integer;
begin
    Result := TStringList.Create;
    P := Pos(SepStr, S);
    while P <> 0 do
    begin
        Result.Add(Copy(S, 1, P-1));
        Delete(S, 1, P-1+Length(SepStr));
        P := Pos(SepStr, S);
    end;
    Result.Add(S);
end;

function StringsToString(SepStr: String; Strs: TStrings;
    GetFileName: Boolean = False): String;
var
    I: Integer;
begin
    Result := '';
    for I := 0 to Strs.Count-1 do

```



```

if not GetFileName then
    Result := Result + SepStr + Strs[I]
else
    Result := Result + SepStr + ExtractFileName(Strs[I]);
    Delete(Result, 1, Length(SepStr));
end;

function Get_ComputerName: String;
var
    iSize: LongWord;
    ComputerName: PChar;
begin
    iSize := MAX_COMPUTERNAME_LENGTH + 1;
    GetMem(ComputerName, iSize);
    GetComputerName(ComputerName, iSize);
    Result := ComputerName;
    FreeMem(ComputerName);
end;

end.

```

服务端主界面程序：

```

unit UT_DL_SRV;

interface

uses
    Windows, Messages, SysUtils, Classes, Controls, Forms, ScktComp,
    StdCtrls, ComCtrls ;

type
    TFM_DL_SRV = class(TForm)
        SrvSocket: TServerSocket;
        sbSRV: TStatusBar;
        pcSRV: TPageControl;
        TabSheet1: TTabSheet;
        UserInfo: TListView;
        procedure SrvSocketGetThread(Sender: TObject;
            ClientSocket: TServerClientWinSocket;
            var SocketThread: TServerClientThread);
        procedure FormCreate(Sender: TObject);

```



```

    procedure FormDestroy(Sender: TObject);
  private
    FileName: TStrings;
  public
    ActiveThreadsCount, BufferSize{以KB为单位}: Integer;
  end;

var
  FM_DL_SRV: TFM_DL_SRV;

implementation

{$R *.dfm}
uses
  UT_SRVTHRD, FunAndProc;

procedure TFM_DL_SRV.FormCreate(Sender: TObject);
var
  Path: String;
begin
  FileName := TStringList.Create;
  Path := ExtractFilePath(ParamStr(0));
  FileName.Add(Path + '\待传输文件1.txt');
  FileName.Add(Path + '\待传输文件2.txt');
  ActiveThreadsCount := 0;
  { 设定数据缓冲区大小为3K }
  BufferSize := 3;
  { 初始化SrvSocket的参数并开始侦听 }
  with SrvSocket do
  begin
    Port := DefaultPort;
    ServerType := stThreadBlocking;
    Open;
  end;
end;

procedure TFM_DL_SRV.FormDestroy(Sender: TObject);
begin
  FreeAndNil(FileName);
end;

procedure TFM_DL_SRV.SrvSocketGetThread(

```

```

    Sender: TObject; ClientSocket: TServerClientWinSocket;
    var SocketThread: TServerClientThread);
begin
    { 建立服务端线程ServerThread, 并传给参数SocketThread }
    SocketThread := TServerThread.Create(
        True, ClientSocket, FileName, BufferSize);
    { 设定该线程结束时自动析构 }
    SocketThread.FreeOnTerminate := True;
    { 启动线程 }
    SocketThread.Resume;
    Inc(ActiveThreadsCount);
    sbSRV.Panels.Items[0].Text := '当前线程数: ' +
        IntToStr(ActiveThreadsCount);
end;

end.

```

以下是线程 TServerThread 的实现代码：

```

unit UT_SRVTHRD;

interface

uses Classes, ScktComp, ComCtrls;

type
    TServerThread = class(TServerClientThread)
    private
        WriteSizes{ 以字节为单位}: Integer; { 向客户端发送文件数据时使用的缓冲区大小 }
        FileName: TStrings; { 文件名列表 }
        FilesStrm: Array of TFileStream; { 文件流数组 }
        FilesLength: Array of Integer; { 文件长度数组 }
        AllFilesLength, FileCurrLength: Integer;
        { 所有文件长度; 已经对某个文件读取了多少长度的数据; 当该长度等于该文件的长度时,
          应该开始读下一个文件 }
        Fileth: Integer; { 当前正在读第几个文件 }
        ListItem: TListItem;
        ErrorRaise: Boolean;
        procedure ListItemAdd;
        procedure ListItemEnd;
        procedure ListItemErr;
        procedure ThreadCountDec;
    end;

```




```

protected
{ TServerClientThread类的执行过程, 相当于普通线程的TThread.Execute}
procedure ClientExecute; override;
public
{重载构造函数, 增加两个参数: AFileName表示要传输的文件名, AWriteSize表示向
  客户端写数据时使用的缓冲区大小}
constructor Create(CreateSuspended: Boolean;
  ASocket: TServerClientWinSocket; AFileName: TStrings;
  AWriteSize: Integer); overload;
destructor Destroy; override;
end;

implementation

uses
  UT_DL_SRV, SysUtils, FunAndProc;

{ ServerThread }

constructor TServerThread.Create(
  CreateSuspended: Boolean; ASocket: TServerClientWinSocket;
  AFileName: TStrings; AWriteSize: Integer);
var
  I: Integer;
begin
  inherited Create(CreateSuspended, ASocket);
  FileName := TStringList.Create;
  FileName.Assign(AFileName);
  WriteSizes := AWriteSize*1024;    { 向客户端写数据时使用的缓冲区大小}
  { 初始化所有变量}
  Fileth := 0;
  FileCurrLength := 0;
  SetLength(FilesStrm, FileName.Count);
  SetLength(FilesLength, FileName.Count);
  AllFilesLength := 0;
  { 创建对应个数的文件流对象}
  for I := 0 to FileName.Count-1 do
  begin
    FilesStrm[I] := TFileStream.Create(
      FileName[I], fmOpenRead or fmShareDenyNone);
    FilesLength[I] := FilesStrm[I].Size;
  end;
end;

```



```

    Inc(AllFilesLength, FilesLength[I]);
end;
ErrorRaise := False;
end;

destructor TServerThread.Destroy;
var
    I: Integer;
begin
    for I := Low(FilesStrm) to High(FilesStrm) do
        FreeAndNil(FilesStrm[I]);
    FreeAndNil(FileName);
    if ErrorRaise then
        { 在一个子线程中对主线程的对象操作时, 应该将这些操作定义在一个过程中, 并使用
          Synchronize来调用这个过程, 以保证操作安全 }
        Synchronize(ListItemErr)
    else
        Synchronize(ListItemEnd);
    Synchronize(ThreadCountDec);
inherited;
end;

procedure TServerThread.ClientExecute;
var
    pStream: TWinSocketStream;
    Buffer: Pointer;
    ReadText, SendText: String;
    I: Integer;
const
    { 读客户端令牌时使用的缓冲区大小, 因为它们都是一些字符串, 所以定义为1024Byte足够了 }
    ReadLen = 1024;
begin
    { 创建连接流对象, 以便和客户端交流 }
    pStream := TWinSocketStream.Create(ClientSocket, 60000);
    try
        { ClientSocket是TServerClientThread类内置的一个对象, 它是和客户端连接的套接字 }
        while (not Terminated) and ClientSocket.Connected do
            begin
                try
                    { 分配读数据缓冲区 }
                    Buffer := AllocMem(ReadLen);

```



```

if pStream.WaitForData(6000) then
begin
  pStream.Read(Buffer^, ReadLen);
  ReadText := PChar(Buffer);
  FreeMem(Buffer);
  { 客户端请求文件名 }
  if ReadText = KEY_Clt[1] then
  begin
    Synchronize(ListItemAdd);
    SendText := KEY_Srv[1] + StringsToString(
      FileNameSepStr, FileName, True);
    { 特别注意SendText后应该加上索引1, 指定Write方法从SendText第一个字符
      开始读, 否则默认从0开始. 那样的话就错了 }
    pStream.Write(SendText[1], Length(SendText)+1);
  end
  { 客户端请求文件长度 }
  else if ReadText = KEY_Clt[2] then
  begin
    SendText := '';
    for I := Low(FilesStrm) to High(FilesStrm) do
      SendText := SendText + FilesLengthSepStr +
        IntToStr(FilesStrm[I].Size);
    Delete(SendText, 1, 1);
    SendText := KEY_Srv[2] + SendText;
    pStream.Write(SendText[1], Length(SendText)+1);
  end
  { 客户端请求发送文件 }
  else if ReadText = KEY_Clt[3] then
  begin
    { 如果当前文件读取完毕, 应该开始读取下一个文件 }
    if FileCurrLength >= FilesLength[Fileth] then
    begin
      Inc(Fileth);
      FileCurrLength := 0;
    end;
    { 分配写入数据缓冲区 }
    Buffer := AllocMem(WriteSizes);
    { 从文件流中读取WriteSizes字节的数据并写入连接流, 最后累加
      FileCurrLength }
    Inc(FileCurrLength, pStream.Write(Buffer^,
      FilesStrm[Fileth].Read(Buffer^, WriteSizes)));
  end
end

```

```

        FreeMem(Buffer);
        { 客户端完成了所有文件的接收, 请求关闭连接}
    end else if ReadText = KEY_Clt[4] then
        Terminate;
    end;
    { 如果发生错误, 则结束线程}
except
    ErrorRaise := True;
    Terminate;
end;
end;
finally
    pStream.Free;
    CltSocket.Close;
end;
end;

procedure TServerThread.ListItemAdd;
begin
    ListItem := FM_DL_SRV.UserInfo.Items.Add;
    ListItem.Caption := DateTimeToStr(Now);
    with ListItem.SubItems do
    begin
        Add(ClientSocket.RemoteHost);
        Add(ClientSocket.RemoteAddress);
        Add(IntToStr(ClientSocket.RemotePort));
        Add(StringsToString(';', FileName));
        Add(IntToStr(FileName.Count));
        Add('传送文件');
    end;
end;

procedure TServerThread.ListItemEnd;
begin
    if ListItem <> nil then with ListItem.SubItems do
        Strings[Count-1] := '传送完毕';
    end;
end;

procedure TServerThread.ListItemErr;
begin
    if ListItem <> nil then with ListItem.SubItems do

```

```
Strings[Count-1] := '传送错误';  
end;  
  
procedure TServerThread.ThreadCountDec;  
begin  
  with FM_DL_SRV do  
  begin  
    Dec(ActiveThreadsCount);  
    sbSRV.Panels.Items[0].Text := '当前线程数:' +  
      IntToStr(ActiveThreadsCount);  
  end;  
end;  
  
end.
```

服务端运行后的界面如图 10-1 所示。

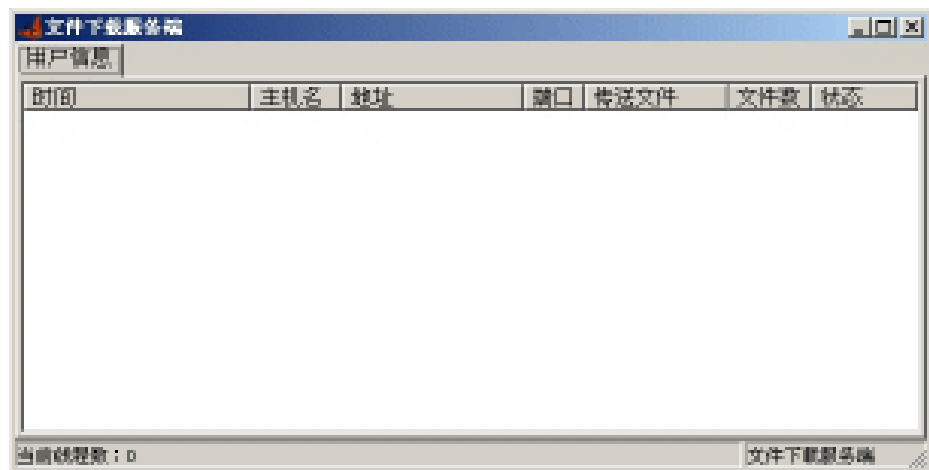


图 10-1 文件下载服务端

小结

本节重点：

- (1) 如何建立线程。
- (2) 使用线程类的 Synchronize 方法来保证资源访问安全。
- (3) 如何使用辅助类 TWinSocketStream 来实现 Socket 通讯。

10.5 实现客户端

根据我们设计的协议，开发出客户端，其代码如下：

主界面：

```

unit UT_DL_CLT;

interface

uses
  Windows, Classes, Forms, ScktComp, StdCtrls, Controls, ComCtrls,
  Gauges, ExtCtrls, UT_CLTTHRD;

type
  TFM_DL_CLT = class(TForm)
    CltSocket: TClientSocket;
    lbNote: TLabel;
    btCommand: TButton;
    ggCopy: TGauge;
    procedure btCommandClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { 客户端线程 }
    ClientThread: TClientThread;
  public
    procedure InitUI(Mode: Word);
  end;

var
  FM_DL_CLT: TFM_DL_CLT;

implementation

uses SysUtils, FunAndProc;

{$R *.dfm}

{ TFM_DL_CLT }
procedure TFM_DL_CLT.FormCreate(Sender: TObject);

```



```
begin
    InitUI(0);
end;

{ InitUI 用来改变主界面组件状态 }
procedure TFM_DL_CLT.InitUI(Mode: Word);
begin
    btCommand.Tag := Mode;
    case Mode of
        0:
            begin
                { 初始化CltSocket }
                with CltSocket do
                    begin
                        Host := Get_ComputerName;
                        Port := DefaultPort;
                    end;
                lbNote.Caption := '请按"开始"开始下载。';
                ggCopy.Progress := 0;
                btCommand.Caption := '开始';
            end;
        1:
            begin
                lbNote.Caption := '正在下载, 请等待.....';
                btCommand.Caption := '取消';
            end;
        2:
            begin
                lbNote.Caption := '下载完毕。';
                ggCopy.Progress := ggCopy.MaxValue;
                btCommand.Caption := '确定';
            end;
    end;
end;

procedure TFM_DL_CLT.btCommandClick(Sender: TObject);
begin
    case TComponent(Sender).Tag of
        0:
            begin
                InitUI(1);
```



```

    { 创建读写线程 }
    ClientThread := TClientThread.Create(
        True, CltSocket, ExtractFilePath(ParamStr(0))+'\'');
    { 线程ClientThread结束时自动销毁 }
    ClientThread.FreeOnTerminate := True;
    { 建立连接 }
    CltSocket.Open;
    { 线程开始运行 }
    ClientThread.Resume;
end;
1:
begin
    ClientThread.Terminate;
    Close;
end;
2:
    Close;
end;
end;
end.

```

以下是线程 TClientThread 的实现代码：

```

unit UT_CLTTHRD;

interface

uses
    Classes, ScktComp;

type
    TClientThread = class(TThread)
    private
        CltSocket: TClientSocket;           { 客户端套接字对象 }
        FileName: TStrings;                 { 服务端传来的文件名 }
        FilesStrm: Array of TFileStream;    { 下载时用来保存文件的流数组 }
        FilesLength: Array of Integer;      { 服务端传来的文件长度 }
        CurrReadSize{ 以字节为单位}: Integer; { 某次读数据操作实际读到的数据 }
        Fileth, AllFilesLength, FileCurrLength: Integer;
        { 当前正在接收第几个文件, 总文件长度, 对当前文件已经接收的数据量 }
    end;

```




```

    GaugeStepRate: Double;           { 进度条每次增加量的百分率}
    ParentDir: String;               { 下载的文件保存到哪个目录}
    procedure Init(LengthText: String);
    { 接收到文件名和文件长度后初始化一系列变量, 为接收文件内容作准备}
    procedure StepProgressToEnd;     { 所有文件下载完毕}
    procedure StepProgress;          { 接收一段文件数据后, 增加进度条的进度}
protected
    procedure Execute; override;
public
    { 重在构造函数, 使它可以传入客户端套接字对象和保存文件的目录}
    constructor Create(CreateSuspended: Boolean;
        ClientSocket: TClientSocket; AParentDir: String); overload;
    destructor Destroy; override;
end;

implementation

uses SysUtils, FunAndProc, UT_DL_CLT;

constructor TClientThread.Create(CreateSuspended: Boolean;
    ClientSocket: TClientSocket; AParentDir: String);
begin
    ParentDir := AParentDir;
    inherited Create(CreateSuspended);
    CltSocket := ClientSocket;
end;

destructor TClientThread.Destroy;
var
    I: Integer;
begin
    for I := Low(FilesStrm) to High(FilesStrm) do
        FreeAndNil(FilesStrm[I]);
    FreeAndNil(FileName);
    inherited;
end;

procedure TClientThread.Execute;
var
    pStream: TWinSocketStream;

```



```

ReadBuffer: Pointer;
ReadText, TaskName, SendText: String;
Start, FileReading: Boolean;
{ 是否已经向服务端发出第一个请求"AskForFileName" ,是否已经准备好开始接收文件数据}
const
  ReadLen = 4*1024;
begin
  Start := False;
  FileReading := False;
  { 建立一个套接字连接流对象}
  pStream := TWinSocketStream.Create(CltSocket.Socket, 60000);
  try
    while (not Terminated) and CltSocket.Active do
      begin

        if not Start then
          begin
            { 发出请求"AskForFileName" }
            SendText := KEY_Clt[1];
            pStream.Write(SendText[1], Length(SendText));
            Start := True;
          end;

          { 分配读取数据缓冲区, 缓冲区大小设置为Windows默认缓冲区大小4*1024Byte}
          ReadBuffer := AllocMem(ReadLen);
          if pStream.WaitForData(6000) then
            begin
              CurrReadSize := pStream.Read(ReadBuffer^, ReadLen);
              if FileReading then
                begin
                  { 读取数据流并保存到文件流}
                  Inc(FileCurrLength, FilesStrm[FileIth].Write(
                    ReadBuffer^, CurrReadSize));
                  { 增加进度}
                  Synchronize(StepProgress);
                  { 如果当前文件的数据已经接收完毕, 则开始接收下一个文件}
                  if FileCurrLength >= FilesLength[FileIth] then
                    begin
                      Inc(FileIth);
                      FileCurrLength := 0;
                    end;
                end;
            end;

```



```

{ 如果所有文件接收完毕, 则向服务端发送"WanttoDisConnect" 并结束线程}
if Fileth = FileName.Count then
begin
    SendText := KEY_Clt[4];
    pStream.Write(SendText[1], Length(SendText));
    Synchronize(StepProgressToEnd);
    Terminate;
end else
{ 接收完第n条数据后向服务端请求第n+1条数据}
begin
    SendText := KEY_Clt[3];
    pStream.Write(SendText[1], Length(SendText));
end;
end else
begin
    ReadText := PChar(ReadBuffer);
    TaskName := Copy(ReadText, 1, Length(KEY_Srv[1]));
    { 如果服务端发来文件名}
    if TaskName = KEY_Srv[1] then
    begin
        Delete(ReadText, 1, Length(KEY_Srv[1]));
        FileName := TStringList.Create;
        FileName.Assign(
            StringToStrings(FileNameSepStr, ReadText));
        SendText := KEY_Clt[2];
        { 向服务端请求文件长度}
        pStream.Write(SendText[1], Length(SendText));
        { 如果服务端发来文件长度}
        end else if TaskName = KEY_Srv[2] then
        begin
            Delete(ReadText, 1, Length(KEY_Srv[1]));
            Init(ReadText);
            SendText := KEY_Clt[3];
            { 向服务端请求发送第一条数据}
            pStream.Write(SendText[1], Length(SendText));
            { 可以开始读文件数据了}
            FileReading := True;
        end;
    end;
end;
end;
FreeMem(ReadBuffer);

```



```

    end;
  finally
    pStream.Free;
    CltSocket.Close;
  end;
end;

procedure TClientThread.Init(LengthText: String);
var
  I: Integer;
  Lengths: TStrings;
begin
  SetLength(FilesStrm, FileName.Count);
  SetLength(FilesLength, FileName.Count);
  Lengths := StringToStrings(FilesLengthSepStr, LengthText);
  Fileth := 0;
  FileCurrLength := 0;
  AllFilesLength := 0;
  { 创建对应个数的文件流对象 }
  for I := 0 to FileName.Count-1 do
  begin
    FileName[I] := ParentDir + '\' + FileName[I];
    FilesStrm[I] := TFileStream.Create(FileName[I], fmCreate);
    FilesLength[I] := StrToInt(Lengths[I]);
    Inc(AllFilesLength, FilesLength[I]);
  end;
  GaugeStepRate := FM_DL_CLT.ggCopy.MaxValue / AllFilesLength;
  FreeAndNil(Lengths);
end;

procedure TClientThread.StepProgress;
begin
  with FM_DL_CLT.ggCopy do
    Progress := Progress + Round(GaugeStepRate*CurrReadSize);
end;

procedure TClientThread.StepProgressToEnd;
begin
  FM_DL_CLT.InitUI(2);
end;

```



end.

客户端运行后的界面如图 10-2 所示。



图 10-2 文件下载客户端

小结

本节重点：

- (1) 如何建立线程。
- (2) 使用线程类的 Synchronize 方法来保证资源访问安全。
- (3) 如何使用辅助类 TWinSocketStream 来实现 Socket 通讯。

10.6 组件封装

大家从上面两节可以看到，服务端和客户端都很容易封装为组件。

对于服务端，它只须知道以下参数就可以提供多文件下载服务了：

- (1) 供下载的文件名。
- (2) 侦听端口。
- (3) 传输文件时使用的缓冲区大小。

对于客户端，它也只须知道以下参数就可以享用文件下载服务了：

- (1) 服务所在主机和侦听端口。
- (2) 下载后的文件保存在什么目录。

对于这些参数，我们花举手之劳即可包装为组件的属性，这样的组件已经能够提供和获取文件下载服务了。如果在发布必要的事件，还可以让用户知道下载过程在任何时刻的状态，如对于服务端来说：当前连接数（即线程数）、某连接的当前状态（开始、结束、出错等）、客户端的信息（主机名、端口等）；对于客户端来说：下载何时开始、当前下载量、下载何时完成等。

大家可以自己动手封装一下，看是不是能够搞定。我封装完成后代码如下：

服务端：

```
unit lxpServerSocketForFiles;  
  
interface
```



```

uses
  Windows, Messages, SysUtils, Classes, ScktComp;

type
  TServerSocketStatus = { 此方法类型用来定义事件OnSocketStatus。}
  procedure (
    Sender: TObject; {TlxpServerSocketForFiles实例本身}
    ClientSocket: TServerClientWinSocket; {和客户端的连接套接字}
    Status: Word; {线程状态标识: 1: 线程启动; 2: 线程正常结束; 3: 线程发生
                  错误结束}
    thisTime: TDateTime {发生该线程状态时的时间}
  ) of object;

  TlxpServerSocketForFiles = class(TServerSocket)
  private
    FFileName: TStrings;
    FActiveThreadsCount, FBufferSize: Integer;
    FOnSocketStatus: TServerSocketStatus;
    { 以下的变量OldOnGetThread和方法NewOnGetThread用来实现对事件OnGetThread的
      嫁接。我们希望在該事件中实现线程自动建立}
    OldOnGetThread: TGetThreadEvent;
    procedure NewOnGetThread(
      Sender: TObject; ClientSocket: TServerClientWinSocket;
      var SocketThread: TServerClientThread);
    procedure SetFileName(const Value: TStrings);
    procedure SetBufferSize(const Value: Integer);
  protected
    procedure Loaded; override;
  public
    property ActiveThreadsCount: Integer read FActiveThreadsCount write
      FActiveThreadsCount;
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    { 发布属性FileName, 用来保存要下载的所有文件名}
    property FileName: TStrings read FFileName write SetFileName;
    { 发布属性BufferSize, 用来指定服务线程发送文件数据时使用的缓冲区大小, 它以KB为
      单位, 并被限制在1~4范围。}
    property BufferSize: Integer read FBufferSize write SetBufferSize;
    { 发布事件OnSocketStatus, 以供组件用户跟踪下载状态}

```



```

    property OnSocketStatus: TServerSocketStatus read FOnSocketStatus
        write FOnSocketStatus;
end;

```

```

procedure Register;

```

```

implementation

```

```

uses

```

```

    ServerThread, FunAndProc;

```

```

procedure Register;

```

```

begin

```

```

    RegisterComponents('lxpbuaa', [TlxpServerSocketForFiles]);

```

```

end;

```

```

{ TlxpServerSocketForFiles }

```

```

constructor TlxpServerSocketForFiles.Create(AOwner: TComponent);

```

```

begin

```

```

    inherited;

```

```

    FFileName := TStringList.Create;

```

```

    FActiveThreadsCount := 0;

```

```

    { 发送文件数据时使用的缓冲区大小被初始化为3K。 }

```

```

    FBufferSize := 3;

```

```

    { 此组件的线程模式默认为stThreadBlocking }

```

```

    ServerType := stThreadBlocking;

```

```

    { 此组件的端口默认为DefaultPort (定义在FunAndProc单元) }

```

```

    Port := DefaultPort;

```

```

end;

```

```

destructor TlxpServerSocketForFiles.Destroy;

```

```

begin

```

```

    FreeAndNil(FFileName);

```

```

    inherited;

```

```

end;

```

{ Loaded是TComponent的一个虚拟方法。当所有组件被创建，并从dfm文件读出数据初始化这些组件实例后，Loaded方法被自动调用。在Loaded中可以进行额外的初始化工作，可以对组件实例的一些成员进行改变、嫁接 }

```

procedure TlxpServerSocketForFiles.Loaded;

```



```

begin
    inherited;
    OldOnGetThread := OnGetThread;
    OnGetThread := NewOnGetThread;
end;

{ 在NewOnGetThread中自动建立服务线程}
procedure TlxpServerSocketForFiles.NewOnGetThread(
    Sender: TObject; ClientSocket: TServerClientWinSocket;
    var SocketThread: TServerClientThread);
begin
    if Assigned(OldOnGetThread) then
        OldOnGetThread(Sender, ClientSocket, SocketThread);
    { 首先执行组件用户在OnGetThread事件中书写的代码, 如果它没有建立服务线程, 那么在组件中自动建立}
    if SocketThread = nil then
    begin
        SocketThread := TServerThread.Create(
            True, ClientSocket, Self, FFileName, FBufferSize);
        SocketThread.FreeOnTerminate := True;
        SocketThread.Resume;
    end;
end;

procedure TlxpServerSocketForFiles.SetBufferSize(const Value: Integer);
begin
    if FBufferSize <> Value then
    begin
        if not (Value in [1..4]) then
            MessageBox(0, 'BufferSize必须在1~4之间。', '提示',
                MB_ICONINFORMATION+MB_OK)
        else FBufferSize := Value;
    end;
end;

procedure TlxpServerSocketForFiles.SetFileName(const Value: TStrings);
var
    I: Integer;
begin
    if Value.Count = 0 then FFileName.Clear
    else

```




```

begin
  for I := 0 to Value.Count-1 do
    if not FileExists(Value[I]) then
      MessageBox(0, PChar('文件:' + Value[I] + '不存在。'), '提示',
        MB_ICONINFORMATION+MB_OK)
    else
      FileName.Assign(Value);
    end;
  end;
end.

```

服务端线程：

```

unit ServerThread;

interface

uses
  Classes, ScktComp, ComCtrls, lxpServerSocketForFiles;

type
  TServerThread = class(TServerClientThread)
  private
    WriteSizes{B}, Fileth: Integer;
    FileName: TStrings;
    FilesStrm: Array of TFileStream;
    FilesLength: Array of Integer;
    AllFilesLength, FileCurrLength: Integer;
    FServerSocket: TlxpServerSocketForFiles;
    { 因为TThread.Synchronize 只能调用无参数过程, 所以定义SetStatus来设置变量
      Status, 然后用Synchronize(ReturnStatus) 根据Status标示的线程状态来激发
      状态事件OnSocketStatus }
    Status: Word;
    { 1: 线程启动 }
    { 2: 线程正常结束 }
    { 3: 线程错误 }
    procedure SetStatus(AStatus: Word);
    procedure ReturnStatus;
  protected
    procedure ClientExecute; override;

```



```

public
  constructor Create(
    CreateSuspended: Boolean; ASocket: TServerClientWinSocket;
    AServerSocket: TlxpServerSocketForFiles; AFileName: TStrings;
    AWriteSize: Integer);overload;
  destructor Destroy; override;
end;

implementation

uses SysUtils, FunAndProc;

{ TServerThread }

procedure TServerThread.ClientExecute;
var
  pStream: TWinSocketStream;
  Buffer: Pointer;
  ReadText, SendText: String;
  I: Integer;
const
  ReadLen = 1024;
begin
  pStream := TWinSocketStream.Create(ClientSocket, 60000);
  try
    while (not Terminated) and ClientSocket.Connected do
      begin
        try
          Buffer := AllocMem(ReadLen);
          if pStream.WaitForData(6000) then
            begin
              pStream.Read(Buffer^, ReadLen);
              ReadText := PChar(Buffer);
              FreeMem(Buffer);
              { 客户请求文件名 }
              if ReadText = KEY_Clt[1] then
                begin
                  SetStatus(1);
                  SendText := KEY_Srv[1] +
                    StringsToString(FileNameSepStr, FileName, True);
                  pStream.Write(SendText[1], Length(SendText)+1);
                end
            end
          else
            continue;
        except
          continue;
        end
      end
    finally
      pStream.Free;
    end
  end

```



```

end
{ 客户请求文件长度}
else if ReadText = KEY_Clt[2] then
begin
    SendText := '';
    for I := Low(FilesStrm) to High(FilesStrm) do
        SendText := SendText + FilesLengthSepStr +
            IntToStr(FilesStrm[I].Size);
    Delete(SendText, 1, 1);
    SendText := KEY_Srv[2] + SendText;
    pStream.Write(SendText[1], Length(SendText)+1);
end
else if ReadText = KEY_Clt[3] then { 请求开始发送文件}
begin
    if FileCurrLength >= FilesLength[Fileth] then
    begin
        Inc(Fileth);
        FileCurrLength := 0;
    end;
    Buffer := AllocMem(WriteSizes);
    Inc(FileCurrLength, pStream.Write(Buffer^,
        FilesStrm[Fileth].Read(Buffer^, WriteSizes)));
    FreeMem(Buffer);
end else if ReadText = KEY_Clt[4] then
begin
    SetStatus(2);
    Terminate;
end;
end;
except
    SetStatus(3);
    Terminate;
end;
end;
finally
    ClientSocket.Close;
    pStream.Free;
end;
end;

```



```

constructor TServerThread.Create(CreateSuspended: Boolean;
  ASocket: TServerClientWinSocket;
  AServerSocket: TlxpServerSocketForFiles;
  AFileName: TStrings; AWriteSize: Integer);
var
  I: Integer;
begin
  inherited Create(CreateSuspended, ASocket);
  { 初始化所有变量 }
  FServerSocket := AServerSocket;
  FileName := TStringList.Create;
  FileName.Assign(AFileName);
  WriteSizes := AWriteSize*1024;
  Fileth := 0;
  FileCurrLength := 0;
  SetLength(FilesStrm, FileName.Count);
  SetLength(FilesLength, FileName.Count);
  AllFilesLength := 0;
  for I := 0 to FileName.Count-1 do
  begin
    FilesStrm[I] := TFileStream.Create(
      FileName[I], fmOpenRead or fmShareDenyNone);
    FilesLength[I] := FilesStrm[I].Size;
    Inc(AllFilesLength, FilesLength[I]);
  end;
end;

destructor TServerThread.Destroy;
var
  I: Integer;
begin
  for I := Low(FilesStrm) to High(FilesStrm) do
    FreeAndNil(FilesStrm[I]);
  FreeAndNil(FileName);
  inherited;
end;

procedure TServerThread.SetStatus(AStatus: Word);
begin
  Status := AStatus;

```



```

{ 在线程安全方法ReturnStatus中激发事件OnSocketStatus}
Synchronize(ReturnStatus);
end;

procedure TServerThread.ReturnStatus;
begin
  with FServerSocket do
    case Status of
      1:
        ActiveThreadsCount := ActiveThreadsCount + 1;
      else
        ActiveThreadsCount := ActiveThreadsCount - 1;
    end;
    { 如果指定了事件处理过程, 就调用它}
    if Assigned(FServerSocket.OnSocketStatus) then
      FServerSocket.OnSocketStatus(
        FServerSocket, ClientSocket, Status, Now);
  end;
end.

```

客户端:

```

unit lxpClientSocketForFiles;

interface

uses
  Windows, Messages, SysUtils, Classes, ScktComp;

type
  TClientSocketStatus = { 此方法类型用来定义客户端事件OnSocketStatus}
  procedure (
    Sender: TObject; { 即TlxpClientSocketForFiles实例本身}
    Status: Word; { 线程状态标识。1: 开始下载文件; 2: 正在下载文件; 3:
                  完成所有文件下载}
    CurrentFilesLength: Integer; { 已经下载的所有文件大小之和, 属性
                                  FilesLength表示所有文件大小之和}
    FileIndex: Integer { 正在下载第几个文件, 从0开始}
  ) of object;

```



```

TlxpClientSocketForFiles = class(TClientsocket)
private
    FSavePath: String;
    FFileName: TStrings;
    FFilesLength: Integer;
    FOnSocketStatus: TClientSocketStatus;
    procedure SetSavePath(const Value: String);
protected
    { 当调用Open方法或者设置属性Active := True时会自动调用虚拟方法DoActivate }
    procedure DoActivate(Value: Boolean); override;
public
    { 发布了运行时属性FileName 和FilesLength }
    property FileName: TStrings read FFileName write FFileName;
    property FilesLength: Integer read FFilesLength write FFilesLength;
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
published
    { 发布属性SavePath, 表示下载的文件保存在哪个目录 }
    property SavePath: String read FSavePath write SetSavePath;
    { 发布事件OnSocketStatus }
    property OnSocketStatus: TClientSocketStatus read FOnSocketStatus
        write FOnSocketStatus;
end;

procedure Register;

implementation

uses ClientThread, FunAndProc;

procedure Register;
begin
    RegisterComponents('lxbuaa', [TlxpClientSocketForFiles]);
end;

{ TlxpClientSocketForFiles }

constructor TlxpClientSocketForFiles.Create(AOwner: TComponent);
begin
    inherited;

```



```

    FFileName := TStringList.Create;
    { 线程模式默认为ctBlocking }
    ClientType := ctBlocking;
    { 端口默认为DefaultPort (定义在FunAndProc单元) }
    Port := DefaultPort;
end;

destructor TlxpClientSocketForFiles.Destroy;
begin
    FreeAndNil(FFileName);
    inherited;
end;

procedure TlxpClientSocketForFiles.DoActivate(Value: Boolean);
var
    ClientThread: TClientThread;
begin
    inherited;
    if Value then
    begin
        ClientThread := TClientThread.Create(True, Self, FSavePath);
        ClientThread.FreeOnTerminate := True;
        ClientThread.Resume;
    end;
end;

procedure TlxpClientSocketForFiles.SetSavePath(const Value: String);
begin
    if (Value = '') or DirectoryExists(Value) then
        FSavePath := Value
    else
        MessageBox(0, PChar('路径:' + Value + '不存在。'), '提示',
            MB_ICONINFORMATION+MB_OK)
end;

end.

```

客户端线程：

```
unit ClientThread;
```



```

interface

uses
  Classes, ScktComp, lxpClientSocketForFiles;

type
  TClientThread = class(TThread)
  private
    CltSocket: TlxpClientSocketForFiles;
    FileName: TStrings;
    FilesStrm: Array of TFileStream;
    FilesLength: Array of Integer;
    CurrReadSize{ 以字节为单位}: Integer;
    Fileth, AllFilesLength, FileCurrLength, AllFileCurrLength: Integer;
    ParentDir: String;
    Status: Word;
    {1: 开始读}
    {2: 正在读}
    {3: 完毕}
    procedure SetStatus(AStatus: Word);
    procedure ReturnStatus;
    procedure Init(LengthText: String);
  protected
    procedure Execute; override;
  public
    constructor Create(CreateSuspended: Boolean;
      ClientSocket: TlxpClientSocketForFiles;
      AParentDir: String);overload;
    destructor Destroy; override;
  end;

implementation

uses SysUtils, FunAndProc;

constructor TClientThread.Create(CreateSuspended: Boolean;
  ClientSocket: TlxpClientSocketForFiles; AParentDir: String);
begin
  ParentDir := AParentDir;
  inherited Create(CreateSuspended);
  CltSocket := ClientSocket;

```




```

end;

destructor TClientThread.Destroy;
var
  I: Integer;
begin
  CltSocket.Close;
  for I := Low(FilesStrm) to High(FilesStrm) do
    FreeAndNil(FilesStrm[I]);
  FreeAndNil(FileName);
  inherited;
end;

procedure TClientThread.Execute;
var
  pStream: TWinSocketStream;
  ReadBuffer: Pointer;
  ReadText, TaskName, SendText: String;
  Start, FileReading: Boolean;
const
  ReadLen = 4*1024;
begin
  Start := False;
  FileReading := False;
  pStream := TWinSocketStream.Create(CltSocket.Socket, 60000);
  try
    while (not Terminated) and CltSocket.Active do
      begin
        if not Start then
          begin
            SendText := KEY_Clt[1];
            pStream.Write(SendText[1], Length(SendText));
            Start := True;
          end;

          ReadBuffer := AllocMem(ReadLen);
          if pStream.WaitForData(6000) then
            begin
              CurrReadSize := pStream.Read(ReadBuffer^, ReadLen);
              if FileReading then

```



```

begin
  Inc(FileCurrLength, FilesStrm[Fileth].Write(ReadBuffer^,
    CurrReadSize));
  Inc(AllFileCurrLength, CurrReadSize);
  SetStatus(2);
  if FileCurrLength >= FilesLength[Fileth] then
  begin
    Inc(Fileth);
    FileCurrLength := 0;
  end;
  if Fileth = FilesName.Count then
  begin
    SendText := KEY_Clt[4];
    pStream.Write(SendText[1], Length(SendText));
    SetStatus(3);
    Terminate;
  end else
  begin
    SendText := KEY_Clt[3];
    pStream.Write(SendText[1], Length(SendText));
  end;
end else
begin
  ReadText := PChar(ReadBuffer);
  TaskName := Copy(ReadText, 1, Length(KEY_Srv[1]));
  if TaskName = KEY_Srv[1] then {文件名}
  begin
    Delete(ReadText, 1, Length(KEY_Srv[1]));
    FileName := TStringList.Create;
    FileName.Assign(StringToStrings(FileNameSepStr,
      ReadText));
    SendText := KEY_Clt[2];
    pStream.Write(SendText[1], Length(SendText));
  end else if TaskName = KEY_Srv[2] then {文件长度}
  begin
    Delete(ReadText, 1, Length(KEY_Srv[1]));
    Init(ReadText);
    SendText := KEY_Clt[3];
    pStream.Write(SendText[1], Length(SendText));
    FileReading := True;
    SetStatus(1);
  end;
end;

```



```

        end;
    end;
    end;
    FreeMem(ReadBuffer);
    end;
finally
    pStream.Free;
    CltSocket.Close;
end;
end;

procedure TClientThread.Init(LengthText: String);
var
    I: Integer;
    Lengths: TStrings;
begin
    SetLength(FilesStrm, FileName.Count);
    SetLength(FilesLength, FileName.Count);
    Lengths := StringToStrings(FilesLengthSepStr, LengthText);
    Fileth := 0;
    FileCurrLength := 0;
    AllFilesLength := 0;
    AllFileCurrLength := 0;
    for I := 0 to FileName.Count-1 do
    begin
        FileName[I] := ParentDir + '\' + FileName[I];
        FilesStrm[I] := TFileStream.Create(FileName[I], fmCreate);
        FilesLength[I] := StrToInt(Lengths[I]);
        Inc(AllFilesLength, FilesLength[I]);
    end;
    FreeAndNil(Lengths);
end;

procedure TClientThread.ReturnStatus;
begin
    if Status = 1 then
        CltSocket.FilesLength := AllFilesLength;
    if Assigned(CltSocket.OnSocketStatus) then
        CltSocket.OnSocketStatus(
            CltSocket, Status, AllFileCurrLength, Fileth);

```



```

end;

procedure TClientThread.SetStatus(AStatus: Word);
begin
    Status := AStatus;
    Synchronize(ReturnStatus);
end;

end.

```

接下来,我们看光盘对应目录的例子(请不要在光盘直接运行,因为客户端组件需要保存下载的文件):

```

unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
    Forms, Dialogs, ScktComp, lxpClientSocketForFiles,
    lxpServerSocketForFiles, StdCtrls, Gauges;

type
    TForm1 = class(TForm)
        lxpServerSocketForFiles1: TlxpServerSocketForFiles;
        lxpClientSocketForFiles1: TlxpClientSocketForFiles;
        Button1: TButton;
        Button2: TButton;
        Gauge1: TGauge;
        Label1: TLabel;
        procedure Button1Click(Sender: TObject);
        procedure FormCreate(Sender: TObject);
        procedure Button2Click(Sender: TObject);
        procedure lxpClientSocketForFiles1SocketStatus(Sender: TObject;
            Status: Word; CurrentFilesLength, FileIndex: Integer);
    private
        Path, DownloadPath: String;
    public
        { Public declarations }
    end;
end;

```



```

var
    Form1: TForm1;

implementation

uses FunAndProc;

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
    Path := ExtractFilePath(ParamStr(0));
    DownloadPath := Path + 'Download';
    {使用Delphi提供的函数ForceDirectories创建目录}
    if not ForceDirectories(DownloadPath) then
    begin
        ShowMessage('请不要在光盘上运行这个例子。');
    end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    with lxpServerSocketForFiles1.FileName do
    begin
        Add(Path + '乖小孩.gif');
        Add(Path + '双胞胎.jpg');
        Add(Path + '浴血长沙——第三次长沙会战守城记.htm');
    end;
    {启动服务端}
    lxpServerSocketForFiles1.Open;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    with lxpClientSocketForFiles1 do
    begin
        SavePath := DownloadPath;
        Host := Get_ComputerName;
        {开始下载}
        Open;
    end;
end;

```



```
end;  
  
{ 在客户端的状态事件中改变进度条的进度}  
procedure TForm1.lxpClientSocketForFiles1.SocketStatus(  
    Sender: TObject; Status: Word; CurrentFilesLength, FileIndex: Integer);  
begin  
    if Gauge1.MaxValue <> lxpClientSocketForFiles1.FilesLength then  
        Gauge1.MaxValue := lxpClientSocketForFiles1.FilesLength;  
    Gauge1.Progress := CurrentFilesLength;  
end;  
  
end.
```

这个例子的运行界面如图 10-3 所示。

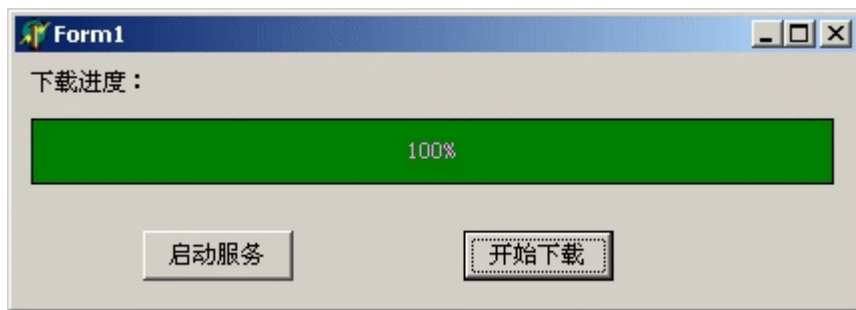


图 10-3 自动下载例子界面

小结

本节的代码演示了如何根据已有应用程序代码来封装组件的过程。封装的要点是：

- (1) 确定组件需要的输入输出信息。
- (2) 输入输出信息可以利用属性、方法或者事件的参数和返回值来获取。具体使用哪些方法，需要根据实际而定。

10.7 自动下载技术在项目中的应用

目前，很多项目采用 C/S 结构来开发。C/S 结构有个重大的缺点就是客户端过于肥大，使得维护极其麻烦。尤其是客户机数量较大的情况下，如果开发人员对客户端程序做了修改，那么就有个更新老程序的问题。如果让开发人员或者技术支持人员跑到每台机子上去更新程序，工作量之大和效率之低是可想而知的。再加上别的原因，所以多层结构和 B/S 结构被越来越多地采用。

在这里，我不想比较 C/S、B/S 和多层结构的优缺点，而打算借助前几节的内容来讨论如何解决



C/S 结构客户端的更新问题。不过，我们的讨论结果显然不仅仅适用于 C/S 结构的问题，对于多层结构，它同样有客户端更新的问题，不过多层结构在这个方面的需要没有 C/S 强烈而已。

很明显，此问题解决的出路在于实现更新自动化，也就是将我们在前面讲述的自动下载技术应用到项目中，这是毋庸置疑的。

如何实现自动更新呢？无非是比较两个值 A 和 B，根据比较结果确定是否需要更新；如果需要更新，则可以使用我们提供的文件自动下载组件 TlxpServerSocketForFiles 和 TlxpClientSocketForFiles 了，我们可以将新文件存放在一台中心计算机（通常可以是数据库服务器、应用程序服务器等），然后从客户计算机下载这些新文件覆盖原来文件；最后运行新的文件即可。

两个组件实现下载已经自动化了，所以上述问题的核心是如何确定 A 和 B 并实现比较。

解决这个核心问题的方法很多，如：

- (1) 在程序中编译进版本信息，比较客户机和中心机上文件的版本信息。
- (2) 比较文件的创建和修改时间等。

以上方法在具体操作时都比较麻烦，而且时间比较方法也不是很可靠，因为许多计算机上的时间都是不准确的。

所以我们在下面介绍一种自定义版本的方法。

将新文件存放在中心机上，并用文件或者数据库记录这些文件的版本信息。如默认为 0，每修改一次版本信息加 1，那么修改 n 次变为 n。并让文件下载服务端运行在中心机上。

在客户机上，需要存放正在使用的程序的版本信息，可以使用文件和注册表。程序每次启动时，要做的第一件事就是连接到中心机，取得指定文件的版本，然后和客户机版本比较，如果客户机版本小，那么表明有新的文件提供下载，于是启动文件下载客户端去下载这些文件。

这样就实现了客户机程序的自动下载功能。

如果你正在为客户机程序的更新而苦恼，那么试试本节讲的方法，相信你很快就能获得跳出苦海的感觉。

